Using Certifying Constraint Solvers for Generating Step-wise Explanations

Ignace Bleukx[1], **Maarten Flippo**[2], **Bart Bogaerts**[1,3], **Emir Demirović**[2], **Tias Guns**[1]
[1]KU Leuven, Department of Computer Science; Leuven.AI, Celestijnenlaan 200A, 3000 Leuven, Belgium
[2]Delft University of Technology, Mekelweg 5, 2628 CD Delft, The Netherlands
[3] Vrije Universiteit Brussel, Dept. of Computer Science, Pleinlaan 2, Brussels, Belgium
ignace.bleukx@kuleuven.be, m.l.flippo@tudelft.nl, bart.bogaerts@kuleuven.be, e.demirovic@tudelft.nl,
tias.guns@kuleuven.be

Abstract

In the field of Explainable Constraint Solving, it is common to explain to a user why a problem is unsatisfiable. A recently proposed method for this is to compute a sequence of explanation steps. Such a step-wise explanation shows individual reasoning steps involving constraints from the original specification, that in the end explain a conflict. However, computing a step-wise explanation is computationally expensive, limiting the scope of problems for which it can be used. We investigate how we can use proofs generated by a constraint solver as a starting point for computing step-wise explanations, instead of computing them step-by-step. More specifically, we define a framework of abstract proofs, in which *both* proofs and step-wise explanations can be represented. We then propose several methods for converting a proof to a step-wise explanation sequence, with special attention to trimming and simplification techniques to keep the sequence and its individual steps small. Our results show our method significantly speeds up the generation of step-wise explanation sequences, while the resulting step-wise explanation has a quality similar to the current state-of-the-art.

Code — https://github.com/ML-KULeuven/Proof2Seq
Extended version — https://arxiv.org/abs/2511.10428

1 Introduction

The ultimate goal of **declarative problem solving** is that users should be able to specify a problem declaratively (i.e., they only need to specify *what* their problem is, not *how* to solve it), and can then use a generic *solver* to find solutions to their problem. Over the last decades, solvers have become increasingly powerful, allowing them to solve complex, real-world problems with thousands of variables and constraints. However, due to this complexity, *explaining to a human* why a solver produced a particular answer is still a challenge.

Explanations Many types of user-oriented explanations have been developed for constraint programming (Gupta, Genc, and O'Sullivan 2021). We focus on explanations for problems that do not admit a solution. More specifically, we focus on methods that explain "why" the problem is unsatisfiable. A popular method is to extract a Minimal Unsatisfiable Subset (MUS) of the input constraints, which allows the user to focus on a problematic subset of the problem specification (Marques-Silva 2010). One issue, however, is that such an MUS can still be very large and hence hard to understand for a user. To better explain the interaction between constraints, and building on old ideas (Sqalli and Freuder 1996), a framework of **step-wise explanations** was introduced by Bogaerts, Gamba, and Guns (2021). In this context, an explanation is a sequence of simple steps, where each step is a simple derivation based on some of the constraints specified by the user. While the framework was originally conceived to explain satisfiable (puzzle) problems, it has also been used later to step-wise explain *unsatisfiability* and *optimality* (Bleukx et al. 2023). A major challenge in the computation of step-wise explanations is how to efficiently find a *short* sequence of *small* explanation steps. Despite novel algorithms designed exactly for this purpose (Gamba, Bogaerts, and Guns 2023), **generating a step-wise explanation remains computationally very expensive**. Indeed, current approaches rely on multiple calls to an NP-oracle for computing even a single explanation step, and many more so for computing the entire sequence. The high runtime of existing approaches limits the general application of step-wise explanation methods. Our goal is to reduce the computational effort for computing a step-wise explanation sequence, by starting from a solver-generated proof instead of constructing it step-by-step.

Proof Logging While combinatorial optimization tools are often used as reliable components in larger systems, their results may not always trustworthy. Indeed, there have been numerous reports of solvers reporting faulty answers (e.g., Brummayer and Biere 2009). The question that naturally arises is: How can we be sure that the answer of a solver is correct? More specifically, if a solver claims that a problem is unsatisfiable or that a solution is optimal, how can we know this is indeed the case? One way to provide such a guarantee is to develop **certifying algorithms** (McConnell et al. 2011), which is also known as **proof logging** in the context of combinatorial optimization. The core idea is that solvers provide a *proof* of correctness of their answer. This approach has been applied very successfully in SAT, with DRAT being the dominant proof logging format (Wetzler, Heule, and Hunt 2014). Recently, proof logging has also found its way to other combinatorial solving formalisms, including MaxSAT (Vandesande, De Wulf, and Bogaerts 2022; Berg et al. 2023, 2024; Vandesande, Coll, and Bogaerts 2026), pseudo-Boolean optimization (Koops et al. 2025; Ihalainen et al. 2026) and CP (Flippo et al. 2024; Gocht, McCreesh, and Nordström 2022).

Goal: Computing Step-wise Explanations from Proofs There are several similarities, but also some crucial differences between *proofs* generated by certifying solvers and a *step-wise explanation sequence*. These differences arise because proofs are designed to be **machine-verifiable**, while step-wise explanations are designed to be **human-interpretable**. Both concepts explain the explanandum through a chain of relatively *simple* steps. However, in proofs, these small steps derive new, potentially complex constraints (e.g., a long clause) that can be reused as part of later proof steps. In contrast, in a step-wise explanation, each step derives a simple *fact*, such as a value assignment of a decision variable by only combining *a few* constraints specified by the user and previously derived facts (Foschini et al. 2026). Hence, solver-generated proofs do not directly provide a user-oriented step-wise explanation.

Another important difference is *how* they are generated. Since proofs are generated efficiently during the solving process, a natural question is whether one can **reuse such a proof to generate a step-wise explanation sequence**. In this paper, we do exactly that and contribute the following:

- We propose a general framework for unsatisfiability proofs that captures both *proofs* and *explanations*;
- we propose a suite of techniques which allow to efficiently extract a step-wise explanation from a proof, with extra attention to minimization techniques that keep the step-wise explanations small (and comprehensible) and;
- we evaluate our approach on several application domains, showing our approach computes step-wise explanations up to 100 times faster, with a limited effect on the length of the explanation and its individual steps.

2 Background

A Constraint Satisfaction Problem (CSP) is a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ with $\mathcal{X}$ a set of integer variables $\mathcal{D}$ a set of domains (one for each variable), and $\mathcal{C}$ a set of constraints (Rossi, van Beek, and Walsh 2006). When the set of variables and domains is clear from the context, we write $\mathcal{C}$ instead of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. Boolean variables are represented as 0-1 integer variables. A constraint in this paper is *any* Boolean expression over $\mathcal{X}$, which maps variable assignments to true or false, and is written as a mathematical expression. E.g., $x + y \leq 3$ and $(x = 3) \vee (y \geq 2)$ both represent a constraint. Variables that occur in the arguments of a constraint are said to be in the constraint's *scope*. We write $scope(c)$ to denote the set of variables occurring in c. The scope trivially generalizes to sets of constraints. An assignment α *satisfies* a constraint when the constraint maps it to true. An assignment that satisfies every constraint in $\mathcal{C}$ is a *solution* of the CSP. If no solution to the CSP exists, it is said to be *unsatisfiable*. The set of solutions to a CSP is written as $sols(\mathcal{C})$. The set of solutions to a CSP, projected to a (sub)-set of variables $\mathcal{V}$ is written as $sols_{\mathcal{V}}(\mathcal{C})$. A set of constraints $\mathcal{C}'$ is *implied by* $\mathcal{C}$ if every solution of $\mathcal{C}$ is also a solution of $\mathcal{C}'$. I.e., $sols(\mathcal{C}) \subseteq sols(\mathcal{C}')$. We write this as $\mathcal{C} \models \mathcal{C}'$, and call $\mathcal{C}'$ a *logical consequence* of $\mathcal{C}$. Now, $\mathcal{C}'$ is a logical consequence of $\mathcal{C}$ if and only if $\mathcal{C} \cup \{\bigvee_{c' \in \mathcal{C}'} \neg c'\}$ is unsatisfiable.

Typically, a user *models* a *user-level* CSP in a constraint modeling system, such as MiniZinc (Nethercote et al. 2007) or CPMpy (Guns 2019), where they can use any Boolean expression as a constraint. This allows the user to focus on the modeling task at hand, without having to take the specifics of the solver into consideration. Next, they choose a general-purpose solver to actually solve the problem. To enable this, the modeling system *transforms* the *user-level* CSP to an equivalent, *solver-level* CSP, using only constraints that are directly supported by the solver (e.g., using flat (global) constraints for CP solvers, or using linear inequalities for MIP solvers (Belov et al. 2016)). During this transformation, the modeling system may introduce *auxiliary variables*. Such variables do not represent an entity in the *user-level* CSP, and a user typically does not care about their value. We assume the transformations implemented in the constraint modeling system do not remove nor introduce new solutions when projected to the original decision variables, i.e., $sols_{\mathcal{X}}(\mathcal{C}) = sols_{\mathcal{X}}(transform(\mathcal{C}))$. When providing the user with an explanation, this should be in terms of *user-level* variables and constraints. E.g., the explanation should not contain any statement regarding auxiliary variables, as the user does not know their meaning.

CP solvers solve a CSP using *propagation-based solving*. That is, the solver *filters* or *propagates* the domains of the decision variables based on the semantics of the constraint. Propagators can have different levels of consistency, with *domain consistent* propagators being the most powerful in terms of propagation strength, often at the cost of a higher runtime of the propagator (Bessiere 2006). We write the propagation function of a solver-level constraint R as f_R.

Throughout this paper, we will often be interested in subsets of constraints that are unsatisfiable. We formalize such subsets using the notion of a Minimal Unsatisfiable Subset (MUS) (Lynce and Marques-Silva 2004).

Definition 1 (Minimal Unsatisfiable Subset)**.** *Given a partitioning of a set of constraints $\mathcal{C}$ into soft constraints $\mathcal{C}_s$ and hard constraints $\mathcal{C}_h$, a Minimal Unsatisfiable Subset (MUS) is a subset $M \subseteq \mathcal{C}_s$ such that $M \cup \mathcal{C}_h$ is unsatisfiable and for any strict subset $M' \subsetneq M$, $M' \cup \mathcal{C}_h$ is satisfiable.*

We write $\mathrm{MUS}(soft : \mathcal{C}_s, hard : \mathcal{C}_h)$ to indicate we compute an MUS for soft constraints $\mathcal{C}_s$ and hard constraints $\mathcal{C}_h$.

3 A Framework for Proofs of Unsatisfiability

As a first contribution, we introduce a unifying framework to describe a proof of why a CSP is unsatisfiable. This framework will allow us to specify differences and similarities between solver-generated proof logs and user-oriented step-wise explanations. We re-interpret and formalize both using the uniform concept of *abstract proofs*. An abstract proof is a sequence of proof steps where each proof step derives new constraints C, using a set of reasons R which are either part of the input CSP or are previously derived in the proof.

Definition 2 (Abstract proof)**.** *Given a CSP* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*, an* abstract proof *is a sequence of pairs* (R_i, C_i) *with* C_i *any set of constraints derived by the proof step, and* R_i *their reasons, another set of constraints. An abstract proof is valid if for each step, the derived constraint is implied by its reasons* ($R_i \models C_i$) *and for each step* i, $R_i \subseteq \mathcal{C} \cup \bigcup_{1 \leq j < i} C_j$.

We will omit curly brackets when the derived set of constraints is a singleton set. An abstract proof of length n proves the unsatisfiability of the CSP if $\perp \in C_n$, i.e., if the last step in the proof derives the "false" literal.

The concept of an abstract proof is, of course, very general, and any proof of unsatisfiability can be represented in it. However, by imposing different restrictions on the content of the proof-steps, an abstract proof instantiates a Deletion Reverse Constraint Propagation (DRCP) prooflog or a step-wise explanation sequence, as we will show next.

3.1 DRCP as abstract proof

If a constraint solver concludes that a solver-level CSP is unsatisfiable, it may produce a proof supporting that conclusion. Here we describe the DRCP format (Flippo et al. 2024), which was designed to represent the behavior of Lazy Clause Generation Constraint Programming solvers.

The smallest unit used in a DRCP proof is the atomic constraint: a statement about the domain of a single variable.

Definition 3 (Atomic Constraint)**.** *An atomic constraint* $\langle x \diamond v \rangle$ *is a comparison between a variable and a constant. I.e.,* $x \in \mathcal{X}$*,* $\diamond \in \{\leq, \geq, =, \neq\}$ *and* $v \in \mathbb{Z}$*.*

Each step in a DRCP proof may be interpreted as an abstract proof-step (R, γ) with γ a *disjunction* of *atomic constraints*, which we will call a *clause*. Based on the content of R, each step in the proof can be categorized as an *inference* or a *nogood-deriving step*.

Definition 4 (Inference proof step)**.** *A proof step* (R, γ) *is an inference step if* γ *is a clause of atomic constraints and* R *is a singleton set with a solver-level constraint.*

Inference proof-steps represent the result of a *constraint propagation*. To allow the proof to be efficiently checked by an external verifier, an inference step also includes which filtering algorithm was used to derive γ from R. An inference is valid with respect to propagator f_R if $f_R(\neg\gamma) = \perp$.

Definition 5 (Nogood deriving step)**.** *A proof step* (R, γ) *is a nogood learning step if* γ *is a clause of atomic constraints, and* R *is a set of previously derived clauses.*

A nogood-deriving step represents a clause (sometimes also called nogood) that is *learned* by the solver, after it found a conflict in a branch of the search-tree (Marques-Silva, Lynce, and Malik 2021; Lecoutre et al. 2009).

Sometimes, not all steps stored in a proof are strictly necessary. This is, for example, the case when a particular step is never used as a *reason* to derive a later step. In a trimmed proof, as defined next, such steps do not occur.

Definition 6 (Trimmed abstract proof)**.** *An abstract proof* $[(R_1, C_1), (R_2, C_2), \ldots, (R_n, \perp)]$ *is trimmed, if each* $c \in C_i$ *appears in at least one reason* R_j *later in the proof.*

Trimming can be implemented efficiently when the proof format stores the set of reasons used to derive a constraint explicitly (Cruz-Filipe et al. 2017), such as in DRCP.

3.2 Step-wise explanations as abstract proofs

The *step-wise explanation framework* was introduced by Bogaerts, Gamba, and Guns (2021) for explaining how to find the unique solution of a logic puzzle using simple derivations and was recently extended to be used in the context of explaining unsatisfiable CSPs (Bleukx et al. 2023). A step-wise explanation of a user-level CSP is an abstract proof where in each proof step (R, C), the derived constraint is a set of atomic constraints (interpreted as a conjunction); and the set of reasons consists of *user-level* constraints, and constraints derived by earlier steps.

Definition 7 (Explanation step)**.** *Given a user-level CSP* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*, an* explanation step *is an abstract proof step* (R_i, C_i) *where* C_i *is a set of atomic constraints.*

Typically, we are not interested in arbitrary explanation steps, but have a preference for *simple* steps. That is, steps in which only a few user-level constraints from the CSP are required to derive the set of new atomic constraints. Most algorithms for computing explanation steps find a minimum set of user constraints to use in a single step.

Example 1. *Figure 2 shows three alternative explanation steps for a Sudoku, explaining why the green cell must be a 4 (i.e.,* $C = x_{3,1} = 4$*.). In the leftmost step, its reasons are* $R = \{\text{DISTINCT}(row_3), \text{DISTINCT}(col_2), x_{2,2} = 4, \ldots\}$

We can build a step-wise explanation sequence by combining such explanation steps. In the literature, several methods for generating such explanation sequences exist (Bogaerts, Gamba, and Guns 2021; Bleukx et al. 2023; Gamba, Bogaerts, and Guns 2023). However, they suffer from scalability issues when many *facts* can or need to be explained, or when many constraints need to be used in a single step.

To summarize, the main difference between DRCP-proof steps and explanation steps are 1) the type of constraints used as reasons (solver-level vs user-level) and 2) the type of constraints derived in a proof step (disjunction vs conjunction). In the next section, we will explore methods that allow us to transform a DRCP-proof into a step-wise explanation.

4 DRCP-proof to Step-wise explanation

We now present our second contribution and propose a set of techniques for processing abstract proofs. By applying these techniques, an abstract proof corresponding to a DRCP proof can be transformed into a step-wise explanation.

4.1 Proof simplification

To transform a DRCP proof into a step-wise explanation, we will need to gradually restrict what is allowed in the proof, and rewrite an abstract proof into an equivalent restricted proof without altering the validity of the proof. For this, we introduce the concept of an abstract P-proof:

Definition 8 (Abstract P-proof)**.** *An abstract P-proof is an abstract proof* $[(R_1, C_1), (R_2, C_2), \ldots, (R_n, C_n)]$ *where property P holds for every step* (R_i, C_i) *in the proof.*

If a proof step (R, C) does not satisfy property P, we remove it from the proof, and update all steps involving C in its reason. That is, any other step (R', C') which uses C in its reason, (i.e., $C \cap R' \neq \emptyset$), can be replaced by the step $((R' \setminus C) \cup R, C')$. In what follows, we illustrate two cases where we can apply this simplification to transform a proof log into a step-wise explanation.

Simplification over auxiliary variables In a step-wise explanation, each step derives atomic constraints about the variables in the user-level CSP. However, as a DRCP proof is a proof of the *solver-level* CSP, its derived constraints may contain auxiliary variables introduced by the modeling system. Using proof simplification, we remove any proof step that derives a constraint over such auxiliary variables. That is, we convert the proof to an abstract P-proof with $P((R, C)) \coloneqq \mathit{scope}(C) \subseteq \mathcal{X}$.

Example 2 (Removing a step with an auxiliary variable)**.** *Consider the following constraint* $a_1 \Rightarrow x + 4 \leq y$ *as a result of transforming the user-level constraint* $(x + 4 \leq y) \vee (y + 6 \leq x)$*. A proof-step involving this constraint is* $(\{a_1 \Rightarrow x + 4 \geq y\}, a_1 = 0 \vee x \neq 3 \vee y \neq 2)$*. Here,* a_1 *was not part of the input CSP and was introduced by the modeling system as part of a transformation. Hence, we remove the step, and in any later proof step, we replace the derived clause with the constraint* $a_1 \Rightarrow x + 4 \leq y$*.*

For abstract proofs certifying unsatisfiability, this is always possible as $\mathit{scope}(\bot) = \emptyset \subseteq \mathcal{X}$. Hence, in the unlikely case where each proof step derives a constraint involving an auxiliary variable, the proof will be simplified to a trivial proof with one step, containing a set of solver-level constraints that derive $\bot$.

Simplification to domain reductions In the step-wise explanation framework, each intermediate result is a set (i.e., a conjunction) of atomic constraints. That is, the derived constraints in each explanation step describe a set of allowed domain values for the variables in the CSP. An abstract proof originating from a DRCP proof may also contain such intermediate results describing a domain. In particular, any clause of atomic constraints over the same variable in the CSP describes a domain. Hence, any step in a proof that derives such a clause can be part of a step-wise explanation. We convert the abstract proof to an abstract P-proof, with $P((R, C)) \coloneqq |\mathit{scope}(C)| \leq 1$. For proofs of unsatisfiability, this is possible as $\mathit{scope}(\bot) = \emptyset$. Thus, in the unlikely case no proof step in the given proof contains a single domain reduction as a derived constraint, the proof can again be simplified to a trivial proof with 1 proof step.

4.2 Solver-level inferences to user-level inferences

Each *inference* step in a solver-generated proof is a *solver-level constraint*, e.g., a linear inequality or a clause, depending on the proof format (Wetzler, Heule, and Hunt 2014). In a step-wise explanation sequence, each step consists of a set of *user-level* constraints. However, multiple solver-level constraints may correspond to one user-level constraint.

By keeping track of which user-level constraint generate a certain solver-level constraint, we can replace any solver-level constraints in the reasons of a proof, if the derived constraint C does not involve an auxiliary variable. Indeed, if the user-level constraint is *stronger* than or equally strong as the solver-level constraint given decision variables $\mathcal{X}$, we can safely replace the solver-level constraint with the user-level constraint, as formalized by the generalized Lemma 1.

Lemma 1 (Validity of user-level constraints in proof steps)**.** *Given a valid proof step* (R, C) *with* $s \in R$*, and* $\mathit{scope}(C) \subseteq \mathcal{X}$*. If* $\mathit{sols}_{\mathcal{X}}(c) \subseteq \mathit{sols}_{\mathcal{X}}(s)$*, then* $(R \setminus \{s\} \cup \{c\}, C)$ *is a valid proof step.*

Example 3 (Replacing a solver-level constraint)**.** *Consider the constraint* $x \neq y$ *as part of the decomposition of the user-level global constraint* $\textsc{Distinct}(x, y, z)$*. The step* $(\{x \neq y\}, (x \neq 1) \vee (y \neq 1))$ *can then safely be replaced with the step* $(\{\textsc{Distinct}(x, y, z)\}, (x \neq 1) \vee (y \neq 1))$*.*

Note, however, that this simplification is only valid if no auxiliary variables occur in the derived constraint C. Indeed, a user-level constraint cannot logically imply a statement about an auxiliary variable introduced by the modeling system. This means that each proof step (R, C) where an auxiliary variable is part of C should be simplified away *before* replacing solver-level constraints with a user-level ones in the proof. This can be done using the proof simplification technique described in Section 4.1.

4.3 Reason minimization

In any abstract proof, including a step-wise explanation, some proof steps may contain more information than strictly necessary. More precisely, some *reasons* may be redundant for deriving a particular derived constraint. In a user-oriented step-wise explanation, such redundant information is unwanted, as it requires more cognitive effort to comprehend. In Algorithm 1, we present a method that minimizes the set of reasons for each step in an abstract proof.

Algorithm 1: MINIMIZEREASONS

```
Input: proof [(R_1, C_1), (R_2, C_2), ..., (R_n, C_n)], mode
Output: trimmed abstract proof with subset-minimal reasons
P ← []; Req ← C_n;
for i = n..1 do                                  ▷ from back to front
    if C_i ∩ Req = ∅ then continue
    if mode = local then cand ← R_i
    if mode = global then cand ← 𝒞 ∪ ⋃_{j<i} C_j
    R'_i ← MUS(soft: cand, hard: ¬C_i)
    Req ← Req ∪ R'_i
    Add step (R'_i, C_i) at the start of P
return P
```

The main idea of the algorithm is to traverse the proof from back to front, and keep a set of *required* (Req) reasons. That is, Req contains at any point the set of reasons that have to be derived by previous steps. If we encounter a step that is not required, it can be discarded from the proof. Once the set of required reasons is empty, we are certain that all steps in the proof have a minimal set of reasons.

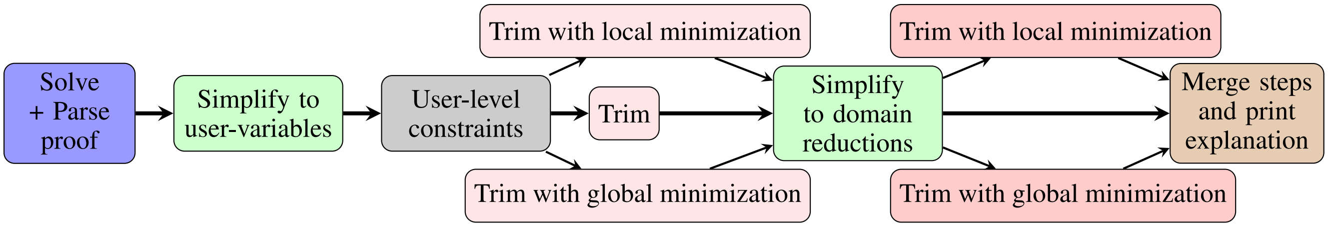

Figure 1: Overview of the pipeline for generating a step-wise explanation sequence from a proof.

For each step, the set of minimal reasons is computed using an MUS algorithm (line 8). Our algorithm is agnostic to the exact method that is used for computing MUSes, and several algorithms for computing such MUSes are available in the literature (Marques-Silva 2010). We use an algorithm that finds an MUS which minimizes the set of user-level constraints (Ignatiev et al. 2015). We consider two versions of minimization algorithm 1: local or global.

With *local minimization*, we only consider the set of given reasons of a proof step as candidate reasons for the minimized step. This mode of the algorithm can be interpreted as a strict *trimming* algorithm as it is agnostic to the inference algorithms used to construct the proof. Indeed, while the solver may require several input constraints or intermediate steps to derive a new constraint, not all of these steps or constraints may be strictly required. This form of trimming can considerably reduce proof size, especially when the propagation algorithms for some constraints are not domain consistent, or when a user-level constraint is *decomposed* into several smaller constraints before solving.

With *global* minimization, we consider *all* user-level constraints in the CSP, and previously derived constraints as candidate reasons. This setting has the advantage that the algorithm is less restricted in its choice of reasons, potentially leading to shorter or simpler proofs. However, this comes at the potential downside of higher computation times of the MUS algorithm. Indeed, because the set of candidates is larger, computing an MUS is conceptually more expensive compared to the local version of the algorithm.

Example 4 (Local and Global Minimization)**.** *Consider the middle proof step in Figure 2. This step needlessly highlights the top-right 3. Local minimization will always exclude the constraint $x_{1,4} = 3$ from its reasons.*

The rightmost proof step cannot be minimized locally as it already uses a subset-minimal set of reasons. However, by considering all possible reasons, we can minimize it to the leftmost proof-step, where we use different Sudoku rules, while deriving the same constraint $\overline{x_{3,1} = 4}$.

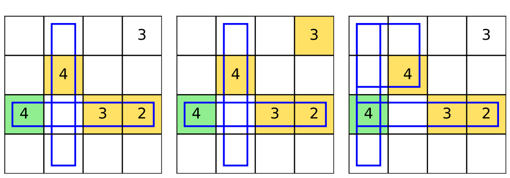

Figure 2: Alternative explanation steps for a 4x4 Sudoku

4.4 Overview of the proposed methods

Figure 1 shows the overview of our proposed pipeline, which allows to transform a DRCP-formatted proof log into a step-wise explanation. The minimal set of processing techniques required is represented as the center line in the diagram.

We first transform the proof log into a user-level proof by removing all proof steps that contain auxiliary variables in the derived constraint using **proof simplification**. Then, we replace all solver-level constraints in the reasons of the proof steps with their user-level constraint it was derived from. We then **trim** the proof to remove any redundant steps. From this smaller proof, we remove all proof steps that derive an intermediate constraint over multiple variables. Finally, we merge proof steps that have the same set of reasons to a single step, and return this step-wise explanation sequence to the user. Optionally, we can apply more elaborate **reason minimization**. Local or global minimization can be applied either directly after the proof is transformed to the user level, or after the proof is simplified to only contain domain reductions. In general, we expect better step-wise explanations when minimizing at the end of the pipeline, as after domain reductions are removed, the proof is already a step-wise explanation. Hence, by applying further minimization, we directly impact the final quality of the step-wise explanation. In contrast, minimizing before the final simplification stage in the pipeline can be faster as each proof step has fewer reasons to consider during the optional, final minimization.

Interestingly, our approach using *global minimization* at the end of the pipeline shares some similarity with the algorithm of Gamba, Bogaerts, and Guns (2023). To compute the next explanation step in the sequence, they iterate over *each* fact to explain and find an *optimal* explanation using an MUS algorithm. Then, across all those facts, the optimal explanation is chosen and added to the sequence as a new step. Crucially, the algorithm of Gamba, Bogaerts, and Guns (2023) does not scale well when many facts can be explained, as it needs to optimally compute explanations for each remaining fact at every step. Hence, their method is more suited to a setting where only few possible facts need explaining, such as the unique solution of a logic puzzle. In contrast, when explaining unsatisfiable models, *any* atomic constraint over any variable is a candidate fact. Our approach using global minimization at the end of the pipeline essentially determines in one go which *fact* should be explained in which part of the sequence, using the order of derived constraints in the proof, and uses an MUS algorithm only to find an optimal set of reasons to explain those facts in turn.

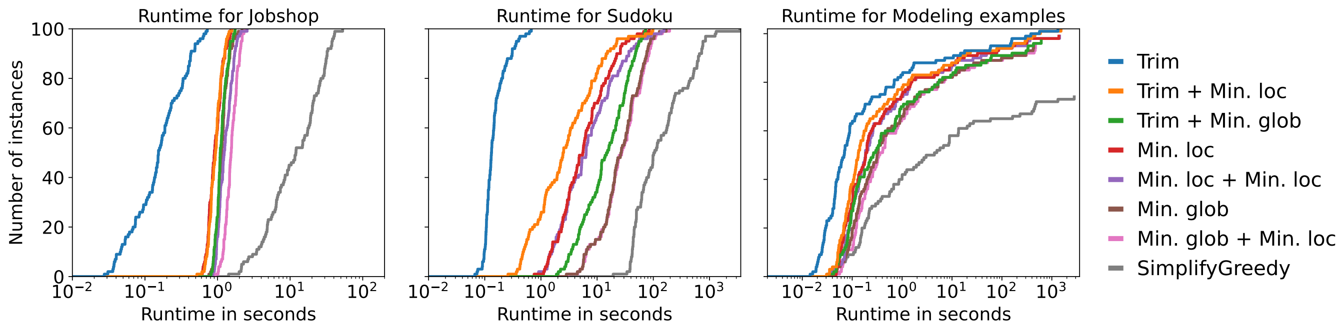

Figure 3: Runtime of all methods for different benchmarks

5 Experimental evaluation

We answer the following experimental questions:

EQ1. How does the runtime of existing approaches compare to our methods?

EQ2. How is the runtime of our approach affected by each combination of proof-minimization?

EQ3. How is the quality of step-wise explanation impacted by each combination of proof-minimization?

Experimental setup We implement our approach in Python 3.11 using the CPMpy (Guns 2019) library v0.9.24. To generate proofs, we use the Pumpkin LCG-solver on commit `d730b05`. For computing MUSes, we employ the SMUS algorithm (Ignatiev et al. 2015), with Exact v2.1.0 (Devriendt 2023) as a SAT-oracle and Gurobi v.12.0.1 as a hitting set solver. As hardware, we used a single core of an Intel(R) Xeon(R) Silver 4514Y and 8GB RAM.

Benchmarks We use three benchmark families, similar to Bleukx et al. (2023). We use 100 unsatisfiable Sudoku instances (**Sudoku**), 100 job-shop instances with a limited makespan in order to make the model unsatisfiable (**Job-shop**), and a diverse set of 102 example models from the CPMpy and CSP-lib repositories (**Modeling examples**), which are altered to contain a modeling error, such as an off-by-one error or swapping a comparison.

Methods under investigation We compare each variant of our approach corresponding to each of the paths in Figure 1: without any optional steps (**Trim**), with local or global minimization at the end of the pipeline (**Trim + Min. loc** or **Trim + Min. glob**), with local or global minimization instead of simple trimming (**Min. Loc** or **Min. Glob**), with and without local minimization at the end of the pipeline (**Min. loc + Min. loc** and **Min. glob + Min. loc**). We do not use any other minimization method in combination with global minimization at the end of the pipeline, as it would ignore the dependencies uncovered by the first minimization anyway. As the current state-of-the-art for computing step-wise explanations of unsatisfiable CSPs, we consider the approach by Bleukx et al. (2023) (**SimplifyGreedy**). We measure the wall-clock time taken by the entire process when using each of these methods and answer the experiment questions.

5.1 EQ1: Comparison of runtime to literature

Figure 3 shows a cumulative distribution plot for all of the methods, for each benchmark. For both the **Sudoku** and **Job-shop** benchmarks, all methods are able to find a step-wise explanation for all instances. However, compared to **SimplifyGreedy**, our approach with minimal post-processing (**Trim**) finds a step-wise explanation ±100x faster. This clearly shows that the overhead of generating the proof during solving once, is negligible compared to the repeated calls to an NP-oracle as done by **SimplifyGreedy**. Indeed, our minimal **Trim** approach does not require *any* NP-reasoning after the solver has produced the proof, as the DRCP format lists the reasons each step explicitly in the proof file. In the **Modeling examples** benchmark, **SimplifyGreedy** is unable to find a step-wise explanation for all instances, and runs out of time (1h) or memory for 29 of the 102 instances. For all benchmarks, even our slowest method (**Min. glob + Min. loc**) finds an explanation sequence at least 10 times faster compared to the state of the art.

5.2 EQ2: Runtime of minimization techniques

As expected, our approach with minimal processing, **Trim**, is overall faster compared to any other method we tested. In general, *local minimization* is faster compared to *global minimization*. This is not surprising as in global minimization, many more candidate reasons may have to be considered to compute the proof step. The runtime difference is especially noticeable in the **Sudoku** benchmark, where, for example, **Trim + Min. loc** finds an explanation sequence around 5-10 times faster compared to **Trim + Min. glob**. Our approach allows us to minimize each proof step either before and/or after simplifying the proof to only contain domain reductions. In general, it is faster to minimize after this simplification. This means the simplification of the proof to only contain domain reductions significantly shortens the proof, compared to before. Indeed, starting with a shorter proof before minimization is faster to process, as for each step, an MUS algorithm has to be called to find a minimal set of reasons. The difference is clear in the **Job-shop** benchmark as scheduling problems are by nature very disjunctive, and hence, intermediate constraints derived in the proof typically do not describe a domain.

Benchmark	Job-shop (100inst)				Sudoku (100 inst)				Modeling examples (73 inst)			
	Sequence length		Max stepsize		Sequence length		Max stepsize		Sequence length		Max stepsize	
	Avg (±std)	Med	Avg (±std)	Med	Avg (±std)	Med	Avg (±std)	Med	Avg (±std)	Med	Avg (±std)	Med
Trim	24.3 (±7.9)	25.0	2.7 (±2.5)	1.0	85.9 (±37.4)	90.0	7.5 (±7.8)	4.0	11.5 (±17.6)	5.0	3.5 (±5.8)	1.0
Trim + Min. loc	2.5 (±4.1)	1.0	1.0 (±0.0)	1.0	82.0 (±36.1)	83.5	4.2 (±4.4)	2.0	8.6 (±15.8)	3.0	2.1 (±2.8)	1.0
Trim + Min. glob	2.2 (±2.9)	1.0	1.0 (±0.0)	1.0	62.6 (±35.9)	57.5	3.4 (±3.6)	1.0	8.6 (±16.0)	3.0	2.0 (±2.6)	1.0
SimplifyGreedy	2.1 (±2.8)	1.0	1.0 (±0.0)	1.0	37.2 (±22.0)	33.0	1.2 (±0.8)	1.0	7.3 (±13.9)	2.0	1.4 (±0.7)	1.0

Table 1: Quality of final explanation sequence, computed on instances finished by all, shown for non-dominated methods.

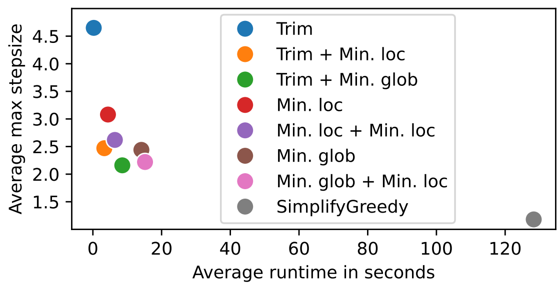

Figure 4: Tradeoff of quality vs runtime on all benchmarks.

5.3 EQ3: explanation quality

Table 1 lists the quality of a step-wise explanation based on its most difficult step in terms of the number of user-level constraints used, and the overall length of the sequence. Note these metrics are correlated, as a sequence using larger steps often needs fewer steps overall to prove the conclusion. Figure 4 plots the runtime and explanation quality of each of the proposed methods. We are interested in methods on the lower left of the plot, i.e., one that is *fast* and computes a *simple* step-wise explanation. Our methods with reason minimization determine a trade-off of explanation quality and runtime. We now compare each of these methods in depth.

Overall, our approach without any minimization (**Trim**) produces the most complex explanation sequences. For example, in the **Job-shop** benchmark, all other methods are able to find explanation sequences with only a single constraint in each step, whereas **Trim** needs on average 2.7. This is also the case for **Sudoku** and **Modeling examples** where **Trim** needs on average 7.5 and 3.5 constraints, respectively.

Sequences produced by **Trim + Min. loc** with local minimization at the end of the pipeline use significantly fewer constraints in their most difficult step. Interestingly, using local minimization instead of simple trimming (**Min. Loc**) also improves the explanation quality for all benchmarks. This means that in most proofs produced by the solver, the set of reasons provided in the proof contains many redundant steps. Indeed, while the solver may require many inference steps to derive a particular proof step, this is not the case after minimizing, as we only require the derived constraint to be logically implied.

Our best overall method is the **Trim + Min. glob** method, which produces explanation sequences of similar quality to those by **SimplifyGreedy**, with a significantly lower runtime. Note that for **Sudoku**, the higher average complexity of the explanation sequence is mainly due to outliers in the data. Indeed, for more than half of the explanation sequences, it produces sequences with only a single constraint in each proof step. Still, on average, **SimplifyGreedy** finds explanation sequences with smaller steps for **Sudoku** and **Modeling examples**. This means there are *facts* that are *easier* to explain compared to those listed in the DRCP proof.

To summarize, while **Trim** finds an explanation very fast, the final quality is worse than with other methods. **SimplifyGreedy** produces the simplest sequences, but at the cost of a significantly higher runtime. Our method **Min. Glob**, computes step-wise explanations that are often of similar quality as **SimplifyGreedy**, but does so in a fraction of the time.

6 Conclusion and Future Work

We proposed a method for computing step-wise explanations from proofs generated by a certifying constraint solver. By representing DRCP formatted proofs and step-wise explanation sequences in a common framework of *abstract proofs*, we have shown how to convert a proof designed to be *machine verifiable* into a step-wise explanation for a *human user*. We have shown several techniques that can remove solver-specific information, such as auxiliary variables and solver-level constraints from the proof, and proposed a set of minimization techniques to further optimize the explanation quality. Different combinations of these minimization techniques result in different tradeoffs between runtime and quality of the final explanation sequence. Our best approach produces explanation sequences of similar quality to the state-of-the-art, with 10 times lower runtime. Our methods can now be applied to larger problems, for which it is unthinkable to use the existing algorithms.

We see two main directions for future work. Firstly, we would like to further reduce the size of the step-wise explanations computed by our approach. For example, by finding shorter proofs (Sidorov et al. 2024) or by adapting the solver behavior to find proofs better suited to transform into a step-wise explanation. Secondly, while the goal of this paper is to find explanations in the already existing format of step-wise explanation sequences, we believe more inspiration from proof logging and proof-theory can be found to improve the actual *explanation*. E.g., while we remove auxiliary variables as they do not fit the step-wise explanation framework, extra variables may be useful to compress the overall proof. However, auxiliary variables can be introduced by the modeling system in many different ways. Therefore, the question of how to integrate auxiliary variables in a user-level explanation is still an open and challenging problem.

Acknowledgments

This work is partially funded by the European Union (ERC, CertiFOX, 101122653; ERC, CHAT-Opt, 01002802 and Europe Research and Innovation program TUPLES, 101070149). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

This work is also partially funded by the Fonds Wetenschappelijk Onderzoek – Vlaanderen (projects G064925N and G070521N).

Maarten Flippo is supported by the project "Towards a Unification of AI-Based Solving Paradigms for Combinatorial Optimisation" (OCENW.M.21.078) of the research programme "Open Competition Domain Science - M" which is financed by the Dutch Research Council (NWO).

References

Belov, G.; Stuckey, P. J.; Tack, G.; and Wallace, M. 2016. Improved Linearization of Constraint Programming Models. In *CP*, volume 9892 of *LNCS*, 49–65. Springer.

Berg, J.; Bogaerts, B.; Nordström, J.; Oertel, A.; Paxian, T.; and Vandesande, D. 2024. Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability. In *CP*, volume 307 of *LIPIcs*, 4:1–4:28. Schloss Dagstuhl.

Berg, J.; Bogaerts, B.; Nordström, J.; Oertel, A.; and Vandesande, D. 2023. Certified Core-Guided MaxSAT Solving. In *CADE*, volume 14132 of *LNCS*, 1–22. Springer.

Bessiere, C. 2006. Constraint Propagation. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, 29–83. Elsevier.

Bleukx, I.; Devriendt, J.; Gamba, E.; Bogaerts, B.; and Guns, T. 2023. Simplifying Step-Wise Explanation Sequences. In *CP*, volume 280 of *LIPIcs*, 11:1–11:20. Schloss Dagstuhl.

Bogaerts, B.; Gamba, E.; and Guns, T. 2021. A framework for step-wise explaining how to solve constraint satisfaction problems. *Artif. Intell.*, 300: 103550.

Brummayer, R.; and Biere, A. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the SMT'09*, 1–5. ISBN 9781605584843.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt, W. A., Jr.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient Certified RAT Verification. In *CADE*, volume 10395 of *LNCS*, 220–236. Springer.

Devriendt, J. 2023. Exact Solver.

Flippo, M.; Sidorov, K.; Marijnissen, I.; Smits, J.; and Demirovic, E. 2024. A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers. In *CP*, volume 307 of *LIPIcs*, 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Foschini, M.; Defresne, M.; Gamba, E.; Bogaerts, B.; and Guns, T. 2026. Preference Elicitation for Step-Wise Explanations in Logic Puzzles. In *Proceedings of The 40th Annual AAAI Conference on Artificial Intelligence*. Accepted for publication.

Gamba, E.; Bogaerts, B.; and Guns, T. 2023. Efficiently Explaining CSPs with Unsatisfiable Subset Optimization. *J. Artif. Intell. Res.*, 78: 709–746.

Gocht, S.; McCreesh, C.; and Nordström, J. 2022. An Auditable Constraint Programming Solver. In *CP*, volume 235 of *LIPIcs*, 25:1–25:18. Schloss Dagstuhl.

Guns, T. 2019. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19.

Gupta, S. D.; Genc, B.; and O'Sullivan, B. 2021. Explanation in Constraint Satisfaction: A Survey. In *IJCAI*, 4400–4407. ijcai.org.

Ignatiev, A.; Previti, A.; Liffiton, M. H.; and Marques-Silva, J. 2015. Smallest MUS Extraction with Minimal Hitting Set Dualization. In *CP*, volume 9255 of *LNCS*, 173–182. Springer.

Ihalainen, H.; Vandesande, D.; Schidler, A.; Berg, J.; Bogaerts, B.; and Järvisalo, M. 2026. Efficient and Reliable Hitting-Set Computations for the Implicit Hitting Set Approach. In *Proceedings of The 40th Annual AAAI Conference on Artificial Intelligence*. Accepted for publication.

Koops, W.; Le Berre, D.; Myreen, M. O.; Nordström, J.; Oertel, A.; Tan, Y. K.; and Vinyals, M. 2025. Practically Feasible Proof Logging for Pseudo-Boolean Optimization. In *CP*, volume 340 of *LIPIcs*, 21:1–21:27. Schloss Dagstuhl.

Lecoutre, C.; Saïs, L.; Tabary, S.; and Vidal, V. 2009. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18): 1592–1614.

Lynce, I.; and Marques-Silva, J. 2004. On Computing Minimum Unsatisfiable Cores. In *SAT*.

Marques-Silva, J. 2010. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *ISMVL*, 9–14. IEEE Computer Society.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 336 of *FAIA*, 133–182. IOS Press.

McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P. 2011. Certifying algorithms. *Comput. Sci. Rev.*, 5(2): 119–161.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, volume 4741 of *LNCS*, 529–543. Springer.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

Sidorov, K.; van der Linden, K.; Correia, G. H. d. A.; de Weerdt, M.; and Demirović, E. 2024. How to discover short, shorter, and the shortest proofs of unsatisfiability: a branch-and-bound approach for resolution proof length minimization. *arXiv preprint arXiv:2411.07955*.

Sqalli, M. H.; and Freuder, E. C. 1996. Inference-Based Constraint Satisfaction Supports Explanation. In *AAAI/IAAI, Vol. 1*, 318–325. AAAI Press / The MIT Press.

Vandesande, D.; Coll, J.; and Bogaerts, B. 2026. Certified Branch-and-Bound MaxSAT Solving. In *Proceedings of The 40th Annual AAAI Conference on Artificial Intelligence*. Accepted for publication.

Vandesande, D.; De Wulf, W.; and Bogaerts, B. 2022. QMaxSATpb: A Certified MaxSAT Solver. In *LPNMR*, volume 13416 of *LNCS*, 429–442. Springer.

Wetzler, N.; Heule, M.; and Hunt, W. A., Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *SAT*, volume 8561 of *LNCS*, 422–429. Springer.