Towards a Certifying Grounder*

Daimy Van Caudenberg
KU Leuven, Leuven, Belgium
daimy.vancaudenberg@kuleuven.be

Alexander Ek
KU Leuven, Leuven, Belgium
ARC Training Centre OPTIMA, Melbourne, Australia
alexander.ek@kuleuven.be

Carlos Cantero
KU Leuven, Leuven, Belgium
carlos.cantero@kuleuven.be

Bart Bogaerts
KU Leuven, Leuven, Belgium
Vrije Universiteit Brussel, Brussels, Belgium
bart.bogaerts@kuleuven.be

Grounding, the translation of high-level theories into equivalent quantifier-free formulas, is a crucial step in declarative solving, yet it has so far escaped the proof-logging revolution. When this grounding step is not certifying, there is no way of knowing that the obtained solutions actually correspond to the original problem specification, resulting in a trust gap. In this paper, we close the trust gap between the user's high-level specification and the solver's low-level input by introducing a novel certifying grounding framework for first-order logic model expansion (FOX) over finite domains. We present CertiFOX, a framework consisting of: (1) a proof format for grounding derivations, (2) GroundFOX, a certifying grounder operating on theories in Grounding Normal Form (GNF)—a new normal form designed for compact, domain-aware grounding—and (3) CheckFOX, an independent proof checker. Our approach guarantees that the grounder's output is equivalent to the input specification, setting the stage for trustworthy end-to-end certified solving pipelines for declarative languages. Experimental evaluation confirms that CERTIFOX is a feasible approach. The GROUNDFOX grounder is broadly comparable with other grounders, and proof checking with CHECKFOX adds overhead within a small constant factor of grounding time.

1 Introduction

The field of combinatorial search and optimization is concerned with solving problems that are often NP-hard. In several subfields, this is done using declarative specifications in a suitable formal language. Over the past decades, we have witnessed impressive improvements in the richness of the languages, the efficiency of solvers, and their industrial applications. Since some of these applications involve high-value and life-affecting decision-making processes (e.g., validating the correctness of plans for space shuttle operation [35], or matching donors and recipients for kidney transplants [31]), it is of utmost importance that the answers produced by the solvers be reliable.

Unfortunately, the reality is different: the constant need for more efficient and advanced algorithms creates an excellent breeding ground for bugs, resulting in numerous reports of bugs in solvers and of solvers outputting faulty answers [9, 10, 27, 12, 1, 18]. The question that naturally arises is how to get stronger guarantees of correctness of a solver. More specifically, if a solver claims that a problem has no solutions, how can we know this is indeed the case? Or if a solver claims a specific solution is optimal, how can we be sure that there are no better solutions?

*Funded by the European Union (ERC, CertiFOX, 101122653). 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.

To appear in EPTCS.

One successful way to achieve such a guarantee is the use of *certifying algorithms* [2, 32]. In the context of combinatorial search, this is also known as proof logging. The idea is that algorithms (in our case for solving decision or optimization problems) should not just output an answer, but also a *certificate* (or *proof*) that this answer is correct. The produced certificate should be sufficiently simple to be verified efficiently by an independent tool, often referred to as the **proof checker**. This checker is a tool of much lesser complexity than the solver. Indeed, the checker does not need any complicated heuristics or search strategies; its sole job is confirming correctness of the derivations made by the solver.

Proof logging has been embraced strongly by the Boolean satisfiability (SAT) community; a wide variety of proof formats have been proposed, the most notable and commonly used one being DRAT [21], and proof logging has been mandatory for solvers participating in the annual SAT competition for years. Inspired by this, proof logging is now seeing adoption in other fields of combinatorial optimization, such as satisfiability modulo theories (SMT) [5], (standard and multi-objective) MaxSAT [38, 26], constraint programming [20, 15], classical planning [14] and answer set programming [3], thereby supporting advanced solving techniques [8, 25]. Proofs of this form are not only useful for guaranteeing the correctness of the solver's answer, but also for debugging and testing solvers [6], and for explainability purposes [7].

In several domains, including constraint programming and answer set programming, users often specify problems in a high-level, human-readable language, which is then translated (by a technique called *grounding*) into a low-level format that a solver can parse. One main limitation of proof logging research so far is that it has focused solely on the solving phase, meaning that the produced certificate only guarantees that the solver's answer is correct with respect to the input it receives, but not that the input itself is correct with respect to the original problem specification, resulting in a *trust gap*.

The goal of this paper is to bridge this gap. Concretely, we approach this problem from a logical perspective and focus on the *model expansion* [34] problems for first-order logic (FOL); in brief, we will write FOX below to refer to first-order logic model expansion. A model expansion problem takes as input a theory in first-order logic, representing domain knowledge, and a structure interpreting a subset of the symbols, representing the concrete instance of the problem at hand. The goal is to expand this input structure to a model of the theory, which represents a solution to the problem.

Typically, a FOX problem is solved in two phases: first, the input theory is *grounded* to an equivalent quantifier-free theory, and then this grounded theory is solved using a suitable solver (e.g., a SAT solver). In practice, state-of-the-art grounders [40, 37, 17] apply sophisticated optimizations and rewrite rules to handle large domains efficiently, making them highly complex and difficult to verify.

Contributions In this paper, we present the foundations of the CERTIFOX ecosystem, a certifying framework for the grounding of FOX specifications. The CERTIFOX ecosystem consists of three main components: the CERTIFOX proof format, the GROUNDFOX certifying grounder, and the CHECKFOX proof checker. We focus on a fragment of first-order logic in which sentences are expressed in *grounding normal form* (GNF), a new normal form designed to exploit domain knowledge to produce compact groundings. The goal of the CERTIFOX ecosystem is to provide a methodology for the certifiable grounding of FOX specifications in GNF, closing the trust gap between the user's specification and the solver's input.

Related Work Proof logging has been studied extensively as an approach to certifying solvers, resulting in many proof formats such as DRAT for SAT [21]; VeriPB for pseudo-Boolean solving [29, 25], MaxSAT [39, 6], and constraint programming [20]; and eDRAT for SMT solving [23]. By contrast, the grounding phase has received little attention from a verification standpoint. The certification of prepro-

cessing steps such as translations is the closest analogue to our work: preprocessing rewrites a formula before solving, much as grounding rewrites a high-level theory before solving. Certifying preprocessing steps has been studied for example in the context of verified translations [19], MaxSAT [24], and quantified Boolean formulas [22]. SMT is particularly relevant, because its preprocessing also encompasses quantifier instantiation, and dedicated proof rules exist to certify these instantiation steps [13, 4, 36]. Unlike grounding, however, SMT instantiation is non-exhaustive—it produces only enough ground instances to satisfy the solver, rather than an equivalent quantifier-free problem.

Our work extends these certifying approaches to include the grounding of first-order logic, closing the trust gap between the user's specification and the solver's input. This is particularly important in the context of high-level modeling languages, where users rely on the grounder to faithfully translate their specifications into a form suitable for solving. IDP-Z3 [11] and CLINGO [16] are high-level modeling systems that perform grounding for FO($\cdot$) and ASP respectively; neither currently produces certificates of correctness for their translations, meaning the final solutions can only be trusted insofar as the grounding process can be trusted. Because both systems are highly complex software systems with many optimizations and rewrite rules, it is difficult to fully verify their correctness, and thus there is a risk of bugs leading to incorrect groundings and, by extension, incorrect final solutions.

2 Preliminaries

A combinatorial search problem asks whether there is an object, within a finite domain, that satisfies all given constraints. For example, given a CNF formula, the problem of finding a satisfying assignment is a combinatorial search problem.

First-Order Logic These preliminaries are based on [40]. A *vocabulary* $\mathcal{V}$ is a set of predicate and function symbols. Each predicate and function symbol has an *arity*, which is the number of arguments it takes. We denote predicate (function) symbols with arity n using P/n (f/n :). A predicate (function) with arity 0 is called a *proposition* (*object*) symbol. We assume that every vocabulary contains the propositions **t** and **f**, representing truth and falsity, respectively.

A *term* is a variable or an n-ary function applied to n terms. An *atom* is an n-ary predicate applied to n terms. If p and q are terms, $p = q$ is also an atom. Next, we define the *formulas*:

- an atom is a formula,
- if ϕ is a formula, then so is $\neg\phi$,
- if ϕ and ψ are formulas, then so are $\phi \Rightarrow \psi$, $\phi \Leftarrow \psi$ and $\phi \Leftrightarrow \psi$,
- if ϕ_i is a formula for all $i \in \{1,\dots,n\}$, then so are $\bigwedge_{i=1}^{n} \phi_i$ and $\bigvee_{i=1}^{n} \phi_i$,
- if ϕ is a formula and x is a variable then $\forall x : \phi$ and $\exists x : \phi$ are formulas.

A *sentence* is a formula without free variables. A *theory* $\mathcal{T}$ over vocabulary $\mathcal{V}$ is a set of sentences over the symbols of $\mathcal{V}$. A *structure* $\mathcal{S}$ over vocabulary $\mathcal{V}$ consists of a domain $\mathcal{D}$ and an interpretation for all symbols in $\mathcal{V}$, where for n-ary predicate symbols, the interpretation is a subset of $\mathcal{D}^n$, for n-ary function symbols, the interpretation is a function from $\mathcal{D}^n$ to $\mathcal{D}$. We assume that the domain $\mathcal{D}$ is finite. A structure $\mathcal{S}$ over $\mathcal{V}$ *expands* a structure $\mathcal{S}'$ over $\mathcal{V}' \subseteq \mathcal{V}$ if they have the same domain and $\mathcal{S}$ agrees with $\mathcal{S}'$ on all symbols in $\mathcal{V}'$ interpreted by $\mathcal{S}'$; this is denoted using $\mathcal{S} \geq_{\mathrm{p}} \mathcal{S}'$ (here the p indicates that the first structure is more precise than the second). A structure $\mathcal{S}$ *satisfies* a sentence ϕ if $\mathcal{S}$ makes ϕ true under the standard semantics of FOL; this is denoted as $\mathcal{S} \models \phi$. A structure $\mathcal{S}$ is a *model* of a theory $\mathcal{T}$ if it satisfies all sentences in $\mathcal{T}$.

Model Expansion A model expansion problem is a combinatorial search problem where a structure over a subset of the vocabulary is given (that is, the *input vocabulary*), and the goal is to find a structure that expands it to the full vocabulary (which includes the *output vocabulary*) and satisfies a given theory. More formally, a First-Order Logic model expansion problem is a tuple $\langle \mathscr{V}, \mathscr{S}_{in}, \mathscr{T} \rangle$ consisting of:

- a vocabulary $\mathscr{V} = \mathscr{V}_{in} \cup \mathscr{V}_{out}$ which is the union of the input and output vocabularies,
- a structure $\mathscr{S}_{in}$ over $\mathscr{V}_{in}$ providing interpretations for input symbols,
- a theory $\mathscr{T}$ over $\mathscr{V}$.

The *FOL model expansion problem* (FOX) consists of finding a structure $\mathscr{S}$ such that:

- $\mathscr{S} \geq_{\mathrm{p}} \mathscr{S}_{in}$ (meaning $\mathscr{S}$ expands $\mathscr{S}_{in}$),
- and $\mathscr{S} \models \mathscr{T}$ (meaning it is a model of $\mathscr{T}$).

We illustrate the flexibility of FOX model specifications with two toy examples.

Example 1. *We now define the pigeonhole problem as a model expansion problem. First, we define the vocabulary* $\mathscr{V} = \{Pig/1, Hol/1, Sit/2\}$*, which contains predicates that allow us to distinguish pigeons from holes, and to assign pigeons to holes. Next, we define the theory:*

$$\mathscr{T} = \left\{ \begin{array}{c} \forall p : Pig(p) \Rightarrow \exists h : Hol(h) \wedge Sit(p,h). \\ \forall p_1, p_2, h : (Hol(h) \wedge Pig(p_1) \wedge Pig(p_2) \wedge p_1 \neq p_2) \Rightarrow \neg Sit(p_1,h) \vee \neg Sit(p_2,h). \end{array} \right\}$$

The sentences ensure that each pigeon sits in a hole and that each hole can contain at most one pigeon. Given the structure $\mathscr{S} = \{\mathscr{D} = \{P_1, P_2, H\}, Pig = \{P_1, P_2\}, Hol = \{H\}\}$ *with fewer holes than pigeons, model expansion of the structure* $\mathscr{S}$ *given theory* $\mathscr{T}$ *fails. Note that for* $\mathscr{S}' = \{\mathscr{D} = \{P_1, P_2, H\}, Pig = \{P_1, P_2\}, Hol = \{P_1, H\}\}$*, model expansion succeeds because the theory does not enforce correct typing.*

Example 2. *The goal is to define a model expansion problem that finds a graph containing a node connected to all other nodes. First, we define the vocabulary* $\mathscr{V} = \mathscr{V}_{in} \cup \mathscr{V}_{out}$*, where the input vocabulary* $\mathscr{V}_{in} = \{Node/1\}$ *contains a predicate that represents nodes, and the output vocabulary contains a predicate for edges (*$\mathscr{V}_{out} = \{Edge/2\}$*). Next, we define the theory* $\mathscr{T}$ *as follows:*

$$\mathscr{T} = \{\exists x : Node(x) \wedge \forall y : Node(y) \wedge y \neq x \Rightarrow Edge(x,y).\}$$

The sentence ensures that there exists a node x such that for all other nodes y, there is an edge from x to y. Model expansion of the structure $\mathscr{S} = \{\mathscr{D} = Node = \{N_1, N_2, N_3, N_4\}\}$ *and theory* $\mathscr{T}$ *will succeed. For instance,* $\mathscr{S}' = \{\mathscr{D} = Node = \{N_1, N_2, N_3, N_4\}$*,* $Edge = \{(N_1, N_2), (N_1, N_3), (N_1, N_4), (N_2, N_3)\}\}$ *is a structure that satisfies the theory; indeed, it contains the node* N_1*, which is connected to all other nodes.*

3 CERTIFOX Ecosystem

The CERTIFOX ecosystem is a novel methodology for the certifiable grounding of FOX specifications. The **GROUNDFOX grounder** takes as input a FOX problem (in a specific normal form defined below), and grounds it to an equivalent CNF formula. During the grounding process, each transformation step is logged in a **CERTIFOX proof**, producing a proof certificate that can be independently verified. The checker then uses the original FOX formula, the grounded CNF formula, and the proof to validate the correctness of the grounding process. This separation of concerns ensures that bugs cannot compromise soundness, as the checker provides an independent verification layer. The architecture is illustrated in Fig. 1, showing the data flow from input formula through grounding and certification.

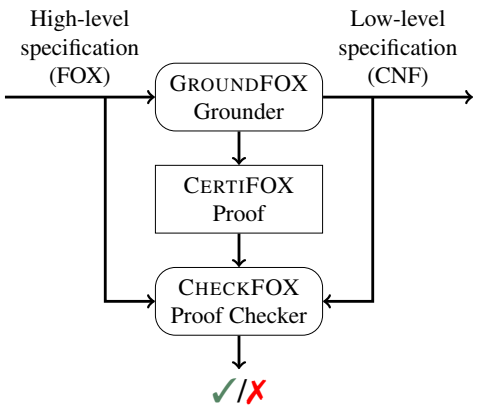

Figure 1: Architecture of the prototype.

3.1 Grounding Normal Form

Our prototype of the CERTIFOX ecosystem is limited to a fragment of FOL in which sentences are expressed in *grounding normal form* (GNF). GNF is a normal form that allows grounding to conjunctive normal form (CNF) after quantifier instantiation and basic simplifications (without the need for the introduction of auxiliary symbols by the grounder). By grounding directly to CNF, users can leverage the power of certifying SAT solvers without needing an additional translation step. This restriction allows us to focus our efforts on certifying the grounding process itself, as opposed to the translation from the grounded formula to conjunctive normal form.

Our normal form is designed to allow for compact grounding by taking advantage of available domain knowledge. This is done by introducing *binary* quantifiers (inspired by [33]), which allow users to define *guards* for the quantifiers. These quantifiers are called *binary* because they take two formulas as arguments: the first is the guard, and the second is the formula being quantified over.

Definition 1 (Binary Quantifiers)**.** *Given formulas φ and ψ, the binary quantifiers used in GNF are:*

$$\forall x[\varphi(x)] : \psi(x) \stackrel{def}{=} \forall x : \varphi(x) \Rightarrow \psi(x)$$
$$\exists x[\varphi(x)] : \psi(x) \stackrel{def}{=} \exists x : \varphi(x) \wedge \psi(x).$$

We call φ a guard for ψ and write $Qx : \psi(x)$ as an abbreviation for $Qx[\mathbf{t}] : \psi(x)$, where $Q \in \{\forall, \exists\}$ and $\mathbf{t}$ is the symbol for truth.

Given a FOX problem $\langle \mathscr{V} = \mathscr{V}_{in} \cup \mathscr{V}_{out}, \mathscr{S}_{in}, \mathscr{T} \rangle$, it is required that guards of the quantifiers in $\mathscr{T}$ only contain symbols from $\mathscr{V}_{in}$, which are interpreted by the user in $\mathscr{S}_{in}$. This allows the grounder to evaluate the guards during the grounding phase, and to skip irrelevant ground instances.

We now introduce the grounding normal form (GNF), which uses these binary quantifiers.

Definition 2 (Grounding Disjunctive Form)**.**

- *If ψ is an atom or a negated atom, then ψ is in grounding disjunctive form.*
- *If ψ and ϕ are in grounding disjunctive form, then $(\psi \vee \phi)$ is in grounding disjunctive form.*
- *If ψ is in grounding disjunctive form and ϕ is a FO-formula all symbols of which are in $\mathscr{V}_{in}$, then $(\exists x[\phi] : \psi)$ is in grounding disjunctive form.*

Definition 3 (Grounding Normal Form)**.**

- *If ψ is in grounding disjunctive form, then ψ is in grounding normal form (GNF).*
- *If ψ and ϕ are in GNF, then $(\phi \wedge \psi)$ is in GNF.*
- *If ψ is in GNF and φ is a FO-formula all symbols of which are in $\mathcal{V}_{in}$, then $(\forall x[\varphi] : \psi)$ is in GNF.*

Note that every FOL formula can be transformed into an equisatisfiable GNF formula. When the quantifiers in the original formula are ordered in a way that is not compatible with GNF, this transformation may require the introduction of auxiliary symbols. If this is not the case, as in Example 1, then the transformation requires only the definition of binary quantifiers and logical equivalences. We illustrate this translation process by transforming the theory from Example 2 into GNF.

Example 3. *We transform the theory from Example 2 into GNF. The original sentence is not in GNF because it contains an existentially quantified universal quantifier. We introduce an auxiliary symbol $ToAll/1$, representing that a node is connected to all other nodes:*

$$\forall x : ToAll(x) \Leftrightarrow \Big(\forall y : y \neq x \Rightarrow Edge(x,y)\Big).$$

We apply standard logical rewrites and use the definition of binary quantifiers to obtain the following equisatisfiable GNF theory:

$$\mathcal{T} = \left\{ \begin{array}{c} \exists x : ToAll(x). \\ \forall x : \forall y\big[y \neq x\big] : Edge(x,y) \vee \neg ToAll(x). \\ \forall x : \exists y\big[y \neq x\big] : \neg Edge(x,y) \vee ToAll(x). \end{array} \right\}$$

3.2 CERTIFOX Proof Format

The goal of a CERTIFOX proof is to certify that a grounder correctly transformed the FOX problem $\langle \mathcal{V}, \mathcal{S}_{in}, \mathcal{T} \rangle$ into an equivalent CNF formula $\mathcal{T}'$, by providing a machine-checkable proof of the equivalence between $\mathcal{T}$ and $\mathcal{T}'$. The proof format is built on a minimal collection of sound, equivalence-preserving rewrite rules, the most important being the instantiation rules for binary quantifiers. We first discuss the rules of the proof system, and then we argue informally that each rule preserves equivalence (given the input structure), which is the key correctness guarantee of CERTIFOX. Next, we describe the syntax of the proof format, and we illustrate it with an example.

CERTIFOX Rules Because the grounder can exploit the known interpretation of input symbols provided by $\mathcal{S}_{in}$, the proof system is designed to preserve a weaker notion of equivalence, called $\mathcal{S}_{in}$-equivalence.

Definition 4 ($\mathcal{S}_{in}$-equivalence)**.** *Given a FOX problem $\langle \mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{out}, \mathcal{S}_{in}, \mathcal{T} \rangle$, two formulas φ and ψ are $\mathcal{S}_{in}$-**equivalent**, written $\varphi \equiv_{\mathcal{S}_{in}} \psi$ if for every structure $\mathcal{S}$ extending $\mathcal{S}_{in}$, it holds that $\mathcal{S}$ is a model of φ if and only if it is a model of ψ (i.e., $\forall \mathcal{S} \geq_{\mathrm{p}} \mathcal{S}_{in} : \mathcal{S} \models \varphi \Leftrightarrow \mathcal{S} \models \psi$). By extension, $\mathcal{T} \equiv_{\mathcal{S}_{in}} \mathcal{T}'$ means that the theories $\mathcal{T}, \mathcal{T}'$ are $\mathcal{S}_{in}$-equivalent, i.e., that they have the same models among the structures that extend $\mathcal{S}_{in}$.*

The correctness guarantee of the CERTIFOX ecosystem is that given an input problem $\langle \mathcal{V}, \mathcal{S}_{in}, \mathcal{T} \rangle$, every rule application preserves $\mathcal{S}_{in}$-equivalence at the theory level, meaning that if $\mathcal{T}$ is the theory before applying the rule and $\mathcal{T}'$ is the theory after applying the rule, then $\mathcal{T} \equiv_{\mathcal{S}_{in}} \mathcal{T}'$. Because most of the rules supported by CERTIFOX are translations of well-known logical equivalences, we will not

$$\text{SNAND}\ \frac{\bigwedge_{i=1\ldots n} \varphi_i}{\bigwedge_{\substack{i=1\ldots n \\ \varphi_i \neq \mathbf{t}}} \varphi_i}\ \varphi_i \neq \mathbf{f} \text{ for all } i \qquad \text{SNAND}\ \frac{\bigwedge_{i=1\ldots n} \varphi_i}{\mathbf{f}}\ \varphi_i = \mathbf{f} \text{ for some } i$$

$$\text{SNOR}\ \frac{\bigvee_{i=1\ldots n} \varphi_i}{\mathbf{t}}\ \varphi_i = \mathbf{t} \text{ for some } i \qquad \text{SNOR}\ \frac{\bigvee_{i=1\ldots n} \varphi_i}{\bigvee_{\substack{i=1\ldots n \\ \varphi_i \neq \mathbf{f}}} \varphi_i}\ \varphi_i \neq \mathbf{t} \text{ for all } i$$

$$\text{EPRED}\ \frac{P(\bar{t}) \qquad \mathscr{S}_{in} \models P(\bar{t})}{\mathbf{t}} \qquad \text{EPRED}\ \frac{P(\bar{t}) \qquad \mathscr{S}_{in} \not\models P(\bar{t})}{\mathbf{f}}$$

$$\text{EPROP}\ \frac{P \qquad \mathscr{S}_{in} \models P}{\mathbf{t}} \qquad \text{EPROP}\ \frac{P \qquad \mathscr{S}_{in} \not\models P}{\mathbf{f}}$$

$$\text{IQ}\ \frac{\forall x[\varphi(x)] : \psi(x)}{\bigwedge_{\substack{v \in \mathscr{D} \\ \mathscr{S}_{in} \models \varphi(v)}} \psi(v)} \qquad \text{IQ}\ \frac{\exists x[\varphi(x)] : \psi(x)}{\bigvee_{\substack{v \in \mathscr{D} \\ \mathscr{S}_{in} \models \varphi(v)}} \psi(v)}$$

$$\text{STN}\ \frac{\neg\mathbf{t}}{\mathbf{f}} \qquad \text{STN}\ \frac{\neg\mathbf{f}}{\mathbf{t}}$$

Table 1: CERTIFOX formula-level proof rules

provide a formal proof of this guarantee for each rule. Instead, we provide an informal argument for each rule, which should be sufficient to convince the reader of the soundness of the proof system.

Tables 1 and 2 contain an overview of the rules of the CERTIFOX proof system; Table 1 contains rules that apply at the level of a single (sub)formula, while Table 2 contains rules that apply at the level of the whole theory. Table 1 contains several rules that allow for the simplification of formulas, such as SNAND, SNOR, and STN. They are basic logical equivalences that hence also preserve $\mathscr{S}_{in}$-equivalence. The rules EPRED and EPROP allow for the interpretation of input predicates and propositions, respectively, by replacing them with $\mathbf{t}$ or $\mathbf{f}$ according to their interpretation in $\mathscr{S}_{in}$. Because this interpretation is fixed, performing this replacement preserves all models among the structures that extend $\mathscr{S}_{in}$; hence it preserves $\mathscr{S}_{in}$-equivalence. The last formula-level rules are the instantiation rules for binary quantifiers (IQ). Concretely, given a formula of the form $Qx[\varphi(x)] : \psi(x)$ with $Q \in \{\forall, \exists\}$, the IQ-rule instantiates the formula only with values v in the domain that satisfy the guard φ according to the input structure $\mathscr{S}_{in}$. This is a crucial optimization, as it allows the grounder to skip irrelevant ground instances. When taking the definition of binary quantifiers into account, it is not hard to see that this rule also preserves $\mathscr{S}_{in}$-equivalence.

Table 2 presents the theory-level rules of the CERTIFOX proof system. Since a theory can be viewed as the conjunction of its formulas, the rules TRIVIAL and UNSAT provide theory-level counterparts of SNAND. Concretely, a formula in the theory that reduces to $\mathbf{t}$ may be removed (TRIVIAL), while one that reduces to $\mathbf{f}$ allows early termination with unsatisfiability (UNSAT). Note that the UNSAT-rule can be used to show the unsatisfiability of the original problem, as it guarantees that there are no models among the structures that extend $\mathscr{S}_{in}$. The SPLITC rule also operates at the theory level and decomposes a conjunctive formula into its conjuncts. It is easy to see that all of these rules preserve $\mathscr{S}_{in}$-equivalence, as they are all based on well-known logical proof rules.

$$\text{TRIVIAL}\ \frac{\mathscr{T} \cup \{\mathbf{t}\}}{\mathscr{T}} \qquad \text{UNSAT}\ \frac{\mathscr{T} \cup \{\mathbf{f}\}}{\text{ABORT}} \qquad \text{SPLITC}\ \frac{\mathscr{T} \cup \{\varphi_1 \wedge \ldots \wedge \varphi_n\}}{\mathscr{T} \cup \{\varphi_1, \ldots, \varphi_n\}}$$

Table 2: CERTIFOX theory-level proof rules

Formula ϕ	**Positions**
$\neg\varphi$	$\langle\phi,0\rangle = \varphi$
$\varphi_1 \circ \varphi_2$	$\langle\phi,0\rangle = \varphi_1$ (lhs), $\langle\phi,1\rangle = \varphi_2$ (rhs)
$\bigcirc(\varphi_0,\ldots,\varphi_n)$	$\langle\phi,i\rangle = \varphi_i$ for $i \in [0,n]$
$Qx[\psi] : \varphi$	$\langle\phi,0\rangle = \psi$ (guard), $\langle\phi,1\rangle = \varphi$ (body)
$P(t_0,\ldots,t_n)$	$\langle\phi,i\rangle = t_i$ for $i \in [0,n]$
$t_1 = t_2$	$\langle\phi,0\rangle = t_1$, $\langle\phi,1\rangle = t_2$
$f(t_0,\ldots,t_n)$	$\langle\phi,i\rangle = t_i$ for $i \in [0,n]$
0-ary predicates and terms	No subformulas or subterms

Table 3: The positioning system for referring to subformulas and subterms, where $\circ \in \{\Leftarrow,\Rightarrow,\Leftrightarrow\}$, $\bigcirc \in \{\wedge,\vee\}$, and $Q \in \{\forall,\exists\}$.

Positioning System Each rule application targets a specific (sub)formula using a custom positioning system built around the formula's syntax tree, ensuring that the proof is unambiguous but brief. The positioning system is crucial for the proof format, as it allows the checker to precisely identify which (sub)formula to apply each rule to. Concretely, a position is a tuple $\langle N,P\rangle$, consisting of a formula name N and a sequence of indices $P = [i_0,i_1,\ldots,i_n]$. The elements $i_j \in \mathbb{N}$ are called indices, and they identify a path from the root of that formula's syntax tree to the subformula to rewrite. A tuple $\langle N,P\rangle$ can also be written as $N[i_0,i_1,\ldots,i_n]$; a tuple with an empty path is written as N, referring to the formula named N itself. Using Table 3, we illustrate how this indexing system can be used to refer to specific subformulas and subterms. In the second formula of Example 3 (assuming it is named 2), the position $\langle 2,[0]\rangle$ refers to the subformula $\forall\big[x \neq y\big] : \text{Edge}(x,y) \vee \neg\text{ToAll}(x)$. The position $\langle 2,[0,0]\rangle$ refers to the guard of the quantifier inside that formula (i.e., $x \neq y$), while the position $\langle 2,[0,1]\rangle$ refers to the body of that quantifier (i.e., $\text{Edge}(x,y) \vee \neg\text{ToAll}(x)$).

CERTIFOX Syntax A CERTIFOX proof consists of three main parts: a header containing version information, the body containing a chronological sequence of applied rewrite rules, and a footer containing identifiers for the obtained grounded formulas. The rewrite rules allow the grounder to log its reasoning in a way that is easy to understand and verify. With the positioning system in place, we can now describe the syntax of the CERTIFOX proof format. The header specifies the version of the proof format and the grounder that produced the proof.

After the header, the body of the proof contains a chronological sequence of applied rewrite rules, which together form a complete record of the transformations performed by the grounder. Each line in the proof corresponds to a single rule application, and is logged as in Listing 1, where each rule application specifies the rule name, the position of the (sub)formula to target and for SPLITC, the names of the new formulas added to the theory after splitting. The footer of the proof contains identifiers for the obtained grounded formulas, which can be used by the checker to verify that the final formula is syntactically identical to the claimed grounding. It is logged using `FINAL IDS : <N>, ... <N>`, listing the names of the formulas that are present in the final grounded theory in the order they appear in the

Listing 1: The syntax of CERTIFOX rule applications.

```
// Rules with @ modify the targeted (sub)formula - SNAND, SNOR, STN, EPRED, IQ, UNSAT
<rule name> @ <position>

// Rules with - delete the targeted formula - TRIVIAL
<rule name> - <position>

// Rules with -> replace the targeted formula with new formulas in the theory - SPLITC
<rule name> <position> -> <position>,<position>,...,<position>
```

theory. If the final grounded theory is empty, this line is logged as `FINAL IDS : -`. An illustration of a syntactically correct CERTIFOX proof is given in Listing 2.

3.3 GROUNDFOX Grounder

The GROUNDFOX grounder is a proof-of-concept implementation that demonstrates the feasibility of certifying the grounding process, serving as a testbed for refining the proof format and supported rewrite rules defined in Section 3.2. It takes as input a FOX problem $\langle \mathcal{V}, \mathcal{S}_{in}, \mathcal{T} \rangle$ with $\mathcal{T}$ in GNF and produces a CNF formula $\mathcal{T}'$ that is $\mathcal{S}_{in}$-equivalent to $\mathcal{T}$, along with a CERTIFOX proof certifying the grounding.

After parsing the input using a custom `ANTLR4` parser and internalizing the theory as a vector of abstract syntax trees, the grounder processes each sentence in order. Each sentence in GNF is a (possibly nested) universal quantifier over a body in Grounding Disjunctive Form. The grounder applies IQ to instantiate the outermost universal quantifier, restricting the domain to elements satisfying the guard. Next, it applies SPLITC to split the resulting conjunction into one top-level formula per domain element. Nested universal quantifiers in the body are handled the same way recursively.

Each instantiated formula is traversed inside-out. These formulas are now all in Grounding Disjunctive Form, so the body is a disjunction of grounding literals, each of which is either a (possibly negated) ground atom or an existentially quantified subformula. First, existential quantifiers are instantiated via IQ into a disjunction of ground literals, after which they are processed identically to the other literals. For each (possibly negated) ground atom, EPRED or EPROP is applied to evaluate the now grounded atom against $\mathcal{S}_{in}$ and replace it with **t** or **f** accordingly; if the atom is negated, STN is applied to flip the resulting truth value. Once all literals are resolved, SNOR collapses the disjunction, short-circuiting to **t** if any literal is true or dropping **f** literals. If at any point a formula simplifies to **f**, the grounder immediately emits UNSAT and terminates without completing the remaining sentences. If a formula simplifies to **t**, it is dropped via TRIVIAL. We illustrate the grounding process with a toy example.

Example 4. *The goal is to ground the FOX problem* $\langle \mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{out}, \mathcal{S}_{in}, \mathcal{T} \rangle$ *with* $\mathcal{V}_{in} = \{P/1, Q/1\}$, $\mathcal{V}_{out} = \{R/1\}$, $\mathcal{T} = \{\forall x[P(x)] : R(x) \vee \exists y : Q(y).\}$ *and* $\mathcal{S}_{in} = \{D = \{1,2,3\}, P = \{2,3\}, Q = \{2\}\}$. *We illustrate the steps applied by the grounder by walking through the obtained proof in Listing 2.* IQ *instantiates the universal quantifier only over elements satisfying the guard* $P(x)$. *After* SPLITC *separates the two formulas, the existential subformula* $\exists y : Q(y)$ *is instantiated over the full domain by a second* IQ *application, and* EPRED *evaluates each ground atom against* $\mathcal{S}_{in}$. *Since* $Q(2)$ *holds,* SNOR *short-circuits the disjunction to* **t**, *before being dropped via* TRIVIAL. *For the remaining formula, the same steps apply. The grounded theory is therefore empty, certified by* `FINAL IDS : -`.

Listing 2: An illustration of a CERTIFOX proof for grounding. Some steps are omitted for brevity.

```
IQ @ 1 // (1)=(R(2) | ? y : Q(y)) & (R(3) | ? y : Q(y))
SPLITC 1 -> 2,3 //(2) = (R(2) | ? y : Q(y)), (3)=(R(3) | ? y : Q(y))
IQ @ 2[1] // (2)=(R(2) | (Q(1) | Q(2) | Q(3)))
EPRED @ 2[1,0] // (2)=(R(2) | (false | Q(2) | Q(3)))
EPRED @ 2[1,1] // (2)=(R(2) | (false | true | Q(3)))
EPRED @ 2[1,2] // (2)=(R(2) | (false | true | false))
SNOR @ 2[1] // (2)=(R(2) | true)
SNOR @ 2 // (2)=true
TRIVIAL - 2 // delete (2)
// repeat EPRED, SNOR, TRIVIAL for (3)
FINAL IDS : - // The grounded theory is empty
```

3.4 CHECKFOX Checker

The CHECKFOX checker is a proof-of-concept implementation that demonstrates the feasibility of independently verifying the correctness of the grounding process using the generated CERTIFOX proofs. It takes as input the original FOX problem $\langle \mathscr{V}, \mathscr{S}_{in}, \mathscr{T} \rangle$, the grounded CNF formula $\mathscr{T}'$ produced by the grounder, and the CERTIFOX proof emitted by the grounder. It validates the correctness of the grounding process by replaying each logged transformation step and checking that the final theory is syntactically identical to the claimed grounding.

Given that all rules of the CERTIFOX proof system preserve $\mathscr{S}_{in}$-equivalence, if the checker successfully verifies the proof, it guarantees that the original theory $\mathscr{T}$ and the grounded theory $\mathscr{T}'$ are $\mathscr{S}_{in}$-equivalent, which is the key correctness guarantee of CERTIFOX. The checker replays the proof sequentially: for each step, it identifies the target subformula by position, applies the specified rule, and rejects immediately if the rule cannot be applied. If all steps succeed, it verifies that the resulting theory is syntactically identical to the claimed grounding, accepting if they match and rejecting otherwise.

Each rule application requires at most a single traversal of the targeted subformula tree, so the checker runs in time linear in the size of the proof (and the formula). Crucially, this means the checker can be kept intentionally simple and small enough that formal verification becomes feasible, which is exactly what makes it a meaningful trust anchor.

4 Experimental Validation

We use the DIRT benchmark suite [30] and select all benchmarks that can be translated into GNF without introducing auxiliary predicates, allowing a fair comparison between the different grounders involved. Concretely, the eight problem sets we use are (1) CommonItem-SAT, (2) CommonItem-UNSAT, (3) CompleteSets-SAT, (4) CompleteSets-UNSAT, (5) GraphColouring, (6) PPM, (7) RamseyNumbers, and (8) stablemarriage. For all of these benchmarks, we created a GNF encoding by directly translating (with minimal changes) an existing (IDP) encoding to our syntax. For one of the problems (stablemarriage), there were multiple IDP encodings; here, we used the encoding on which IDP-Z3 was the most efficient.

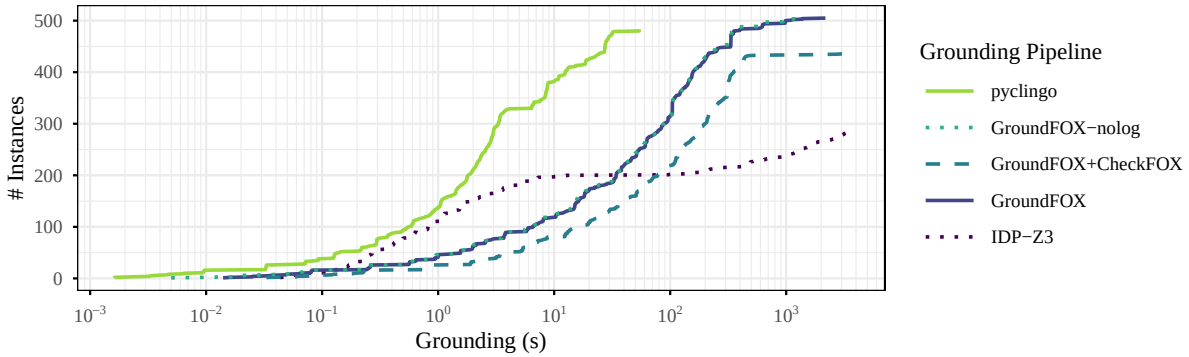

Figure 2: Number of instances y with individual completion time $\leq x$ across grounding pipelines.

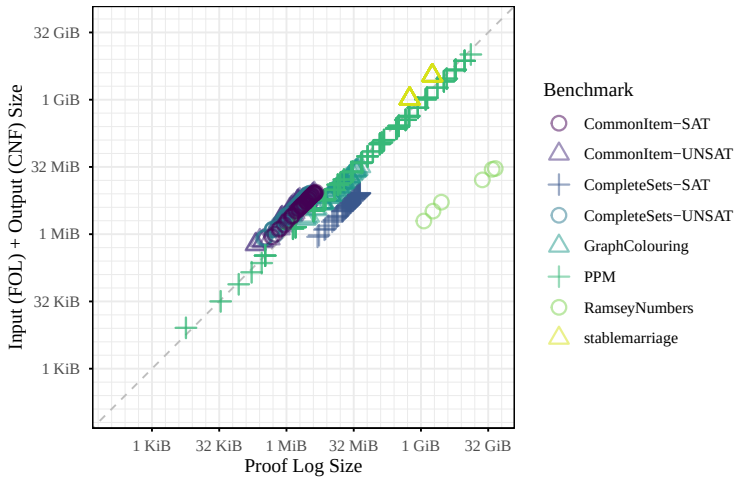

Figure 3: Scatter plot of proof sizes for the different benchmarks.

4.1 Setup

Because of time constraints, we could not compare against all available grounders. Instead, we compare only against IDP-Z3 (0.12.0) and PYCLINGO (5.8.0), which are state-of-the-art solving frameworks for problems expressed in FO(·) and ASP, respectively. These frameworks follow a ground-and-solve approach and therefore contain highly optimized grounders. We consider three versions of our tool:[1] the grounder (GROUNDFOX 0.1.0) without proof logging, the grounder with proof logging, the grounder with proof logging and checking (CHECKFOX 0.1.0).

Experiments were conducted on an Intel Xeon Platinum 8468 CPU cluster running Rocky Linux 8.10. Each run was executed as a single-core job with 8 GiB RAM, with multiple runs scheduled concurrently across the cluster. Each instance was grounded and then solved with a 1-hour time limit.

[1]The source code is available at `https://gitlab.kuleuven.be/krr/software/groundfox-checkfox`

4.2 Runtime Results

First, we compare the runtime of the grounders. Fig. 2 shows grounding runtimes for the benchmark instances across grounding pipelines. Compared to PYCLINGO and IDP-Z3, the performance of GROUNDFOX grounder is respectable overall, although it is outperformed by PYCLINGO on all benchmarks and by IDP-Z3 on about half. IDP-Z3 plateaus after solving around 200 instances: it performs particularly well on the PPM benchmark but struggles with the remaining benchmarks. We believe this is because the PPM instances involve a small typed integer domain with arithmetic constraints that the underlying Z3 SMT solver handles efficiently.

The plot also shows that the overhead of proof logging is minimal in GROUNDFOX, and the overhead of checking is within a factor of 2–3 in most cases. Interestingly, GROUNDFOX was able to ground most instances given the memory and time limits. In particular, PYCLINGO runs out of memory on 30 instances of PPM and runs out of time on 2 instances of stablemarriage, while GROUNDFOX grounds all stablemarriage instances successfully and runs out of memory on only 4 instances of PPM.

We note that the tools are not equally affected by the memory and time constraints. As mentioned, out of a total of 515 instances, PYCLINGO hit the time limit on 2 and the memory limit on 30, while IDP-Z3 timed out on 117 and ran out of memory on 112 instances. GROUNDFOX (with and without proof logging) timed out on 6 and ran out of memory on only 4 instances. CHECKFOX timed out on 3 instances and ran out of memory on 66 of the 505 successfully grounded instances, making it the most memory-intensive step; addressing this remains future work.

4.3 Proof Size Results

We are also interested in the size of the generated proofs, as this is an important factor for the feasibility of proof checking. Fig. 3 presents a scatter plot of the proof sizes for the different benchmarks. This figure shows that the proof sizes vary widely across benchmarks, with some proofs being relatively small (a few hundred lines) and others being quite large (millions of lines). This variation in proof size makes sense given the large variation in domains. When comparing the proof size against the size of the input and obtained output, we see that the RamseyNumbers proofs are larger. In the GNF encoding for the RamseyNumbers problems, no guards are used for the quantifiers, which in combination with the deeply nested quantifiers leads to an exponential number of formulas to simplify in the proof. This stands in contrast to the other problem families: IDP is a typed system supporting multi-sorted logic, and types in an IDP encoding translate directly into guards on the quantifiers in the corresponding GNF encoding. RamseyNumbers is the only problem family whose IDP encoding does not use types, and therefore yields unguarded quantifiers in the GNF encoding.

5 Conclusion

We presented CERTIFOX, a certifying framework for grounding FOX problems, comprising GROUNDFOX, a certifying grounder for GNF problems producing compact groundings via domain knowledge; the CERTIFOX proof format; and CHECKFOX, an independent proof checker. Experiments show that GROUNDFOX is broadly comparable to IDP-Z3 and PYCLINGO, with negligible proof-logging overhead, while CHECKFOX verifies proofs within a constant factor of grounding time. Future work includes broader language coverage (arbitrary FOL sentences, cardinality constraints), a formally verified checker, and seamless integration into high-level solving pipelines, as well as extending the proof system with more complex reasoning.

References

[1] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel & Peter Nightingale (2018): *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*, *Lecture Notes in Computer Science* 11008, Springer, pp. 727–736, doi:10.1007/978-3-319-98334-9_46. Available at `https://doi.org/10.1007/978-3-319-98334-9_46`.

[2] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah & Pascal Schweitzer (2011): *An Introduction to Certifying Algorithms*. *it Inf. Technol.* 53(6), pp. 287–293, doi:10.1524/itit.2011.0655. Available at `https://doi.org/10.1524/itit.2011.0655`.

[3] Mario Alviano, Carmine Dodaro, Johannes Klaus Fichte, Markus Hecher, Tobias Philipp & Jakob Rath (2019): *Inconsistency Proofs for ASP: The ASP - DRUPE Format*. *Theory Pract. Log. Program.* 19(5-6), pp. 891–907, doi:10.1017/S1471068419000255. Available at `https://doi.org/10.1017/S1471068419000255`.

[4] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury & Pascal Fontaine (2020): *Scalable Fine-Grained Proofs for Formula Processing*. *J. Autom. Reason.* 64(3), pp. 485–510, doi:10.1007/s10817-018-09502-y. Available at `https://doi.org/10.1007/s10817-018-09502-y`.

[5] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli & Clark W. Barrett (2022): *Flexible Proof Production in an Industrial-Strength SMT Solver*. In Jasmin Blanchette, Laura Kovács & Dirk Pattinson, editors: *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, *Lecture Notes in Computer Science* 13385, Springer, pp. 15–35, doi:10.1007/978-3-031-10769-6_3. Available at `https://doi.org/10.1007/978-3-031-10769-6_3`.

[6] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel & Dieter Vandesande (2023): *Certified Core-Guided MaxSAT Solving*. In Brigitte Pientka & Cesare Tinelli, editors: *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, *Lecture Notes in Computer Science* 14132, Springer, pp. 1–22, doi:10.1007/978-3-031-38499-8_1. Available at `https://doi.org/10.1007/978-3-031-38499-8_1`.

[7] Ignace Bleukx, Maarten Flippo, Bart Bogaerts, Emir Demirovic & Tias Guns (2026): *Using Certifying Constraint Solvers for Generating Step-wise Explanations*. In Koenig et al. [28], pp. 14192–14200, doi:10.1609/AAAI.V40I17.38432. Available at `https://doi.org/10.1609/aaai.v40i17.38432`.

[8] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh & Jakob Nordström (2023): *Certified Dominance and Symmetry Breaking for Combinatorial Optimisation*. *J. Artif. Intell. Res.* 77, pp. 1539–1589, doi:10.1613/jair.1.14296. Available at `https://doi.org/10.1613/jair.1.14296`.

[9] Robert Brummayer & Armin Biere (2009): *Fuzzing and Delta-Debugging SMT Solvers*. In Ofer Strichman Bruno Dutertre, editor: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, Association for Computing Machinery, New York, NY, USA, p. 15, doi:10.1145/1670412.1670413. Available at `https://doi.org/10.1145/1670412.1670413`.

[10] Robert Brummayer, Florian Lonsing & Armin Biere (2010): *Automated Testing and Debugging of SAT and QBF Solvers*. In Ofer Strichman & Stefan Szeider, editors: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, *Lecture Notes in Computer Science* 6175, Springer, pp. 44–57, doi:10.1007/978-3-642-14186-7_6. Available at `https://doi.org/10.1007/978-3-642-14186-7_6`.

[11] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens & Marc Denecker (2022): *IDP-Z3: a reasoning engine for FO(·)*. *CoRR* abs/2202.00343. arXiv:2202.00343.

[12] William J. Cook, Thorsten Koch, Daniel E. Steffy & Kati Wolter (2013): *A hybrid branch-and-bound approach for exact rational mixed-integer programming*. *Math. Program. Comput.* 5(3), pp. 305–344, doi:10.1007/s12532-013-0055-6. Available at `https://doi.org/10.1007/s12532-013-0055-6`.

[13] David Déharbe, Pascal Fontaine & Bruno Woltzenlogel Paleo (2011): *Quantifier Inference Rules for SMT proofs*. In Pascal Fontaine & Aaron Stump, editors: *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving, Wrocław, Poland, August 1, 2011*, pp. 33–39. Available at `https://pxtp2011.loria.fr/PxTP2011.pdf#page=37`.

[14] Salomé Eriksson, Gabriele Röger & Malte Helmert (2017): *Unsolvability Certificates for Classical Planning*. In Laura Barbulescu, Jeremy Frank, Mausam & Stephen F. Smith, editors: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, AAAI Press, pp. 88–97. Available at `https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15734`.

[15] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits & Emir Demirovic (2024): *A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers*. In Paul Shaw, editor: *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, *LIPIcs* 307, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 11:1–11:20, doi:10.4230/LIPICS.CP.2024.11. Available at `https://doi.org/10.4230/LIPIcs.CP.2024.11`.

[16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Philipp Wanko (2016): *Theory Solving Made Easy with Clingo 5*. In Manuel Carro, Andy King, Neda Saeedloei & Marina De Vos, editors: *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, *OASIcs* 52, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 2:1–2:15, doi:10.4230/OASIcs.ICLP.2016.2. Available at `https://doi.org/10.4230/OASIcs.ICLP.2016.2`.

[17] Martin Gebser, Torsten Schaub & Sven Thiele (2007): *GrinGo : A New Grounder for Answer Set Programming*. In Chitta Baral, Gerhard Brewka & John S. Schlipf, editors: *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, *Lecture Notes in Computer Science* 4483, Springer, pp. 266–271, doi:10.1007/978-3-540-72200-7_24. Available at `https://doi.org/10.1007/978-3-540-72200-7_24`.

[18] Xavier Gillard, Pierre Schaus & Yves Deville (2019): *SolverCheck: Declarative Testing of Constraints*. In Thomas Schiex & Simon de Givry, editors: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, *Lecture Notes in Computer Science* 11802, Springer, pp. 565–582, doi:10.1007/978-3-030-30048-7_33. Available at `https://doi.org/10.1007/978-3-030-30048-7_33`.

[19] Stephan Gocht, Ruben Martins, Jakob Nordström & Andy Oertel (2022): *Certified CNF Translations for Pseudo-Boolean Solving*. In Kuldeep S. Meel & Ofer Strichman, editors: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, *LIPIcs* 236, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:25, doi:10.4230/LIPIcs.SAT.2022.16. Available at `https://doi.org/10.4230/LIPIcs.SAT.2022.16`.

[20] Stephan Gocht, Ciaran McCreesh & Jakob Nordström (2022): *An Auditable Constraint Programming Solver*. In Christine Solnon, editor: *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, *LIPIcs* 235, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:18, doi:10.4230/LIPIcs.CP.2022.25. Available at `https://doi.org/10.4230/LIPIcs.CP.2022.25`.

[21] Marijn Heule, Warren A. Hunt, Jr. & Nathan Wetzler (2013): *Trimming while checking clausal proofs*. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, IEEE, pp. 181–188. Available at `https://ieeexplore.ieee.org/document/6679408/`.

[22] Marijn Heule, Martina Seidl & Armin Biere (2014): *A Unified Proof System for QBF Preprocessing*. In Stéphane Demri, Deepak Kapur & Christoph Weidenbach, editors: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, *Lecture Notes in Computer Science* 8562, Springer, pp. 91–106, doi:10.1007/978-3-319-08587-6_7. Available at `https://doi.org/10.1007/978-3-319-08587-6_7`.

[23] S. Hitarth, Cayden R. Codel, Hanna Lachnitt & Bruno Dutertre (2024): *Extending DRAT to SMT*. In Nina Narodytska & Philipp Rümmer, editors: *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, IEEE, pp. 1–11, doi:10.34727/2024/ISBN.978-3-85448-065-5_8. Available at https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_8.

[24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen & Jakob Nordström (2024): *Certified MaxSAT Preprocessing*. In Christoph Benzmüller, Marijn J. H. Heule & Renate A. Schmidt, editors: *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, *Lecture Notes in Computer Science* 14739, Springer, pp. 396–418, doi:10.1007/978-3-031-63498-7_24. Available at https://doi.org/10.1007/978-3-031-63498-7_24.

[25] Hannes Ihalainen, Dieter Vandesande, André Schidler, Jeremias Berg, Bart Bogaerts & Matti Järvisalo (2026): *Efficient and Reliable Hitting-Set Computations for the Implicit Hitting Set Approach*. In Koenig et al. [28], pp. 14251–14260, doi:10.1609/AAAI.V40I17.38439. Available at https://doi.org/10.1609/aaai.v40i17.38439.

[26] Christoph Jabs, Jeremias Berg, Bart Bogaerts & Matti Järvisalo (2025): *Certifying Pareto Optimality in Multi-Objective Maximum Satisfiability*. In Arie Gurfinkel & Marijn Heule, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*, *Lecture Notes in Computer Science* 15697, Springer, pp. 108–129, doi:10.1007/978-3-031-90653-4_6. Available at https://doi.org/10.1007/978-3-031-90653-4_6.

[27] Matti Järvisalo, Marijn Heule & Armin Biere (2012): *Inprocessing Rules*. In Bernhard Gramlich, Dale Miller & Uli Sattler, editors: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, *Lecture Notes in Computer Science* 7364, Springer, pp. 355–370, doi:10.1007/978-3-642-31365-3_28. Available at https://doi.org/10.1007/978-3-642-31365-3_28.

[28] Sven Koenig, Chad Jenkins & Matthew E. Taylor, editors (2026): *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*. AAAI Press. Available at https://aaai.org/proceeding/aaai-40-2026/.

[29] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan & Marc Vinyals (2025): *Practically Feasible Proof Logging for Pseudo-Boolean Optimization*. In Maria Garcia de la Banda, editor: *31st International Conference on Principles and Practice of Constraint Programming, CP 2025, August 10-15, 2025, Glasgow, Scotland*, *LIPIcs* 340, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 21:1–21:27, doi:10.4230/LIPICS.CP.2025.21. Available at https://doi.org/10.4230/LIPIcs.CP.2025.21.

[30] Lucas Van Laer, Simon Vandevelde & Joost Vennekens (2025): *DIRT: a Literature-Based Benchmark Suite for Grounders*. In Giovanni Casini, Besik Dundua & Temur Kutsia, editors: *Logics in Artificial Intelligence - 19th European Conference, JELIA 2025, Kutaisi, Georgia, September 1-4, 2025, Proceedings, Part I*, *Lecture Notes in Computer Science* 16093, Springer, pp. 343–356, doi:10.1007/978-3-032-04587-4_21. Available at https://doi.org/10.1007/978-3-032-04587-4_21.

[31] David F. Manlove & Gregg O'Malley (2014): *Paired and Altruistic Kidney Donation in the UK: Algorithms and Experimentation*. *ACM J. Exp. Algorithmics* 19(1), doi:10.1145/2670129. Available at https://doi.org/10.1145/2670129.

[32] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher & Pascal Schweitzer (2011): *Certifying algorithms*. *Comput. Sci. Rev.* 5(2), pp. 119–161, doi:10.1016/j.cosrev.2010.09.009. Available at https://doi.org/10.1016/j.cosrev.2010.09.009.

[33] Michaelis Michael & A. V. Townsend (1995): *Binary Quantification Systems*. *Notre Dame Journal of Formal Logic* 36(3), pp. 382–395, doi:10.1305/ndjfl/1040149354.

[34] David G. Mitchell & Eugenia Ternovska (2005): *A Framework for Representing and Solving NP Search Problems*. In Manuela M. Veloso & Subbarao Kambhampati, editors: *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, AAAI Press / The MIT Press, pp. 430–435. Available at http://www.aaai.org/Library/AAAI/2005/aaai05-068.php.

[35] Monica L. Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson & Matthew Barry (2001): *An A-Prolog Decision Support System for the Space Shuttle*. In I. V. Ramakrishnan, editor: *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings*, *Lecture Notes in Computer Science* 1990, Springer, pp. 169–183, doi:10.1007/3-540-45241-9_12. Available at https://doi.org/10.1007/3-540-45241-9_12.

[36] Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett & Cesare Tinelli (2022): *Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language*. In: *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 65–74, doi:10.34727/2022/isbn.978-3-85448-053-2_10. Available at https://repositum.tuwien.at/bitstream/20.500.12708/81324/1/Noetzli-2022-Reconstructing%20Fine-Grained%20Proofs%20of%20Rewrites%20Using%20a%20Domai...-vor.pdf.

[37] Tommi Syrjänen (1998): *Implementation of Local Grounding for Logic Programs with Stable Model Semantics*. Technical Report B18, Helsinki University of Technology, Finland.

[38] Dieter Vandesande, Jordi Coll & Bart Bogaerts (2026): *Certified Branch-and-Bound MaxSAT Solving*. In Koenig et al. [28], pp. 14342–14351, doi:10.1609/AAAI.V40I17.38449. Available at https://doi.org/10.1609/aaai.v40i17.38449.

[39] Dieter Vandesande, Wolf De Wulf & Bart Bogaerts (2022): *QMaxSATpb: A Certified MaxSAT Solver*. In Georg Gottlob, Daniela Inclezan & Marco Maratea, editors: *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, *Lecture Notes in Computer Science* 13416, Springer, pp. 429–442, doi:10.1007/978-3-031-15707-3_33. Available at https://doi.org/10.1007/978-3-031-15707-3_33.

[40] Johan Wittocx, Maarten Mariën & Marc Denecker (2010): *Grounding FO and FO(ID) with Bounds*. *J. Artif. Intell. Res.* 38, pp. 223–269, doi:10.1613/jair.2980. Available at https://doi.org/10.1613/jair.2980.