

Certified Knowledge Compilation with Application to Verified Model Counting

Randal E. Bryant   




Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Wojciech Nawrocki   

Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Jeremy Avigad   

Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule   

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Computing many useful properties of Boolean formulas, such as their weighted or unweighted model count, is intractable on general representations. It can become tractable when formulas are expressed in a special form, such as the decision-decomposable, negation normal form (dec-DNNF). *Knowledge compilation* is the process of converting a formula into such a form. Unfortunately existing knowledge compilers provide no guarantee that their output correctly represents the original formula, and therefore they cannot validate a model count, or any other computed value.

We present *Partitioned-Operation Graphs* (POGs), a form that can encode all of the representations used by existing knowledge compilers. We have designed CPOG, a framework that can express proofs of equivalence between a POG and a Boolean formula in conjunctive normal form (CNF).

We have developed a program that generates POG representations from dec-DNNF graphs produced by the state-of-the-art knowledge compiler D4, as well as checkable CPOG proofs certifying that the output POGs are equivalent to the input CNF formulas. Our toolchain for generating and verifying POGs scales to all but the largest graphs produced by D4 for formulas from a recent model counting competition. Additionally, we have developed a formally verified CPOG checker and model counter for POGs in the Lean 4 proof assistant. In doing so, we proved the soundness of our proof framework. These programs comprise the first formally verified toolchain for weighted and unweighted model counting.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Propositional model counting, Proof checking

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.6

Supplementary Material *DataPaper (Paper Supplement)*:

<https://doi.org/10.5281/zenodo.7966174>

Funding *Randal E. Bryant*: Supported by NSF grant CCF-2108521.

Wojciech Nawrocki: Hoskinson Center for Formal Mathematics

Jeremy Avigad: Hoskinson Center for Formal Mathematics

Marijn J. H. Heule: Supported by NSF grant CCF-2108521.

1 Introduction

Given a Boolean formula ϕ , modern Boolean satisfiability (SAT) solvers can find an assignment satisfying ϕ or generate a proof that no such assignment exists. They have applications across a variety of domains including computational mathematics, combinatorial optimization, and the formal verification of hardware, software, and security protocols. Some applications, however, require going beyond Boolean satisfiability. For example, the *model counting problem*



© Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023).

Editors: Meena Mahajan and Friedrich Slivovsky; Article No. 6; pp. 6:1–6:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

requires computing the number of satisfying assignments of a formula, including in cases where there are far too many to enumerate individually. Model counting has applications in artificial intelligence, computer security, and statistical sampling. There are also many useful extensions of standard model counting, including *weighted model counting*, where a weight is defined for each possible assignment, and the goal becomes to compute the sum of the weights of the satisfying assignments.

Model counting is a challenging problem – more challenging than the already NP-hard Boolean satisfiability. Several tractable variants of Boolean satisfiability, including 2-SAT, become intractable when the goal is to count models and not just determine satisfiability [28]. Nonetheless, a number of model counters that scale to very large formulas have been developed, as witnessed by the progress in recent model counting competitions.

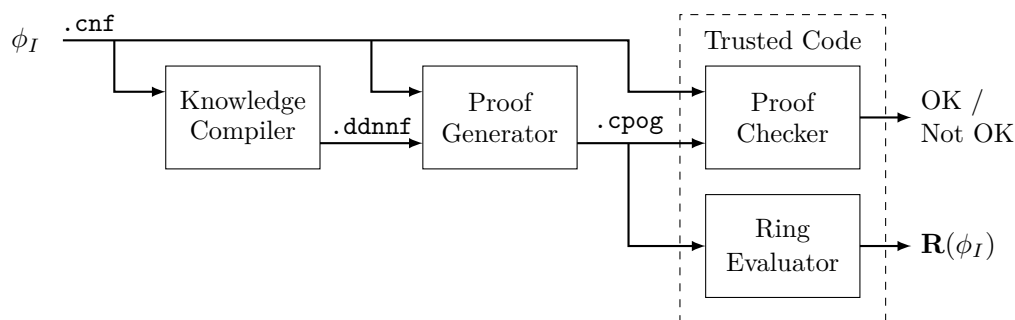
One approach to model counting, known as *knowledge compilation*, transforms the formula into a structured form for which model counting is straightforward. For example, the *deterministic, decomposable, negation normal form* (det-DNNF) introduced by Darwiche [7, 8], as well as the more restricted *decision-DNNF* (dec-DNNF) [17, 24], represent a Boolean formula as a directed acyclic graph, with terminal nodes labeled by Boolean variables and their complements, and with each nonterminal node labeled by a Boolean And or Or operation. Restrictions are placed on the structure of the graph (described in Section 5) such that a count of the models can be computed by a single bottom-up traversal. Kimmig et al. present a very general *algebraic model counting* [19] framework describing properties of Boolean functions that can be efficiently computed from a det-DNNF representation. These include standard and weighted model counting, and much more.

One shortcoming of existing knowledge compilers is that they have no generally accepted way to validate that the compiled representation is logically equivalent to the original formula. By contrast, all modern SAT solvers can generate checkable proofs when they encounter unsatisfiable formulas. The guarantee provided by a checkable certificate of correctness enables users of SAT solvers to fully trust their results. Experience has also shown that being able to generate proofs allow SAT solver developers to quickly detect and diagnose bugs in their programs. This, in turn, has led to more reliable SAT solvers.

This paper introduces *Partitioned-Operation Graphs* (POGs), a form that can encode all of the representations produced by current knowledge compilers. The CPOG (for “certified” POG) file format then captures both the structure of a POG and a checkable proof of its logical equivalence to a Boolean formula in conjunctive normal form (CNF). A CPOG proof consists of a sequence of clause addition and deletion steps, based on an extended resolution proof system [27]. We establish a set of conditions that, when satisfied by a CPOG file, guarantees that it encodes a well-formed POG and provides a valid equivalence proof.

Figure 1 illustrates our certifying knowledge compilation and model counting toolchain. Starting with input formula ϕ_I , the D4 knowledge compiler [20] generates a dec-DNNF representation, and the *proof generator* uses this to generate a CPOG file. The *proof checker* verifies the equivalence of the CNF and CPOG representations. The *ring evaluator* computes a standard or weighted model count from the POG representation. As the dashed box in Figure 1 indicates, this toolchain moves the root of trust away from the complex and highly optimized knowledge compiler to a relatively simple checker and evaluator. Importantly, the proof generator need not be trusted – its errors will be caught by the proof checker.

To ensure soundness of the abstract CPOG proof system, as well as correctness of its concrete implementation, we formally verified the proof system as well as versions of the proof checker and ring evaluator in the Lean 4 proof assistant [11]. Running these two programs on a CPOG file gives strong assurance that the proof and the model count are correct. Our



■ **Figure 1** Certifying toolchain. The output of a standard knowledge compiler is converted into a combined graph/proof (CPOG) which can be independently checked and evaluated.

experience with developing a formally verified proof checker has shown that, even within the well-understood framework of extended resolution, it can be challenging to formulate a full set of requirements that guarantee soundness. In fact, our efforts to formally verify our proof framework exposed subtle conditions that we had to impose on our partitioned sum rule.

We evaluate our toolchain using benchmark formulas from the 2022 standard and weighted model competitions. Our tools handle all but the largest dec-DNNF graphs generated by D4. We also measure the benefits of several optimizations as well as the relative performance of the verified checker with one designed for high performance and capacity.

We also show that our tools can provide end-to-end verification of formulas that have been transformed by an equivalence-preserving preprocessor. That is, verification is based on the original formula, and so proof checking certifies correct operation of the preprocessor, the knowledge compiler, and the proof generator.

We have developed an online supplement to this paper [3] that includes a worked example, more details on the algorithms, and extensive experimental results.

2 Related Work

Generating proofs of unsatisfiability in SAT solvers has a long tradition [31] and has become widely accepted due to the formulation of clausal proof systems for which proofs can readily be generated and efficiently checked [15, 30]. A number of formally verified checkers have been developed within different verification frameworks [6, 14, 21, 26]. The associated proofs add clauses while preserving satisfiability until the empty clause is derived. Our work builds on the well-established technology and tools associated with clausal proof systems, but we require features not found in proofs of unsatisfiability. Our proofs construct a new propositional formula, and we must verify that it is equivalent to the input formula. This requires verifying additional proof steps, including clause deletion steps, and subtle invariants, as described in Sections 7 and 10.

Capelli, Lagniez, and Marquis developed a knowledge compiler that generates a certificate in a proof system that is itself based on dec-DNNF [4, 5]. Their CD4 program, a modified version of D4, generates annotations to the compiled representation, providing information about how the compiled version relates to the input clauses. It also generates a file of clausal proof steps in the DRAT format [30]. Completing the certification involves running two different checkers on the annotated dec-DNNF graph and the DRAT file. Although the authors make informal arguments regarding the soundness of their frameworks, these do not provide strong levels of assurance. Indeed, we have identified a weakness in their methodology

due to an invalid assumption about the guarantees provided by DRAT-TRIM, the program it uses to check the DRAT file. This weakness is *exploitable*: their framework can be “spoofed” into accepting an incorrect compilation.

In more detail, CD4 emits a sequence of clauses R that includes the conflict clauses that arose during a top-down processing of the input clauses. Given input formula ϕ_I , their first task is to check whether $\phi_I \Rightarrow R$, i.e., that any assignment that satisfies ϕ_I also satisfies each of the clauses in R . They then base other parts of their proof on that property and use a separate program to perform a series of additional checks. They use DRAT-TRIM to prove the implication, checking that each clause in R satisfies the *reverse asymmetric tautology* (RAT) property with respect to the preceding clauses [15, 18]. Adding a RAT clause C to a set of clauses maintains satisfiability, but it does not necessarily preserve models. As an example, consider the following formulas:

$$\begin{aligned}\phi_1: & (x_1 \vee x_3) \\ \phi_2: & (x_1 \vee x_3) \quad \wedge \quad (x_2 \vee \bar{x}_3)\end{aligned}$$

Clearly, these two formulas are not equivalent – ϕ_1 has six models, while ϕ_2 has four. In particular, ϕ_1 allows arbitrary assignments to variable x_2 . Critically, however, the second clause of ϕ_2 is RAT with respect to ϕ_1 – any satisfying assignment to ϕ_1 can be transformed into one that also satisfies ϕ_2 by setting x_2 to 1, while keeping the values for other variables fixed.

This weakness would allow a buggy (or malicious) version of CD4 to spoof the checking framework. Given formula ϕ_1 as input, it could produce a compiled result, including annotations, based on ϕ_2 and also include the second clause of ϕ_2 in R . The check with DRAT-TRIM would pass, as would the other tests performed by their checker. We have confirmed this possibility with their compiler and checker.¹

This weakness can be corrected by restricting DRAT-TRIM to only add clauses that obey the stronger *reverse unit propagation* (RUP) property [13, 29]. We have added a command-line argument to DRAT-TRIM that enforces this restriction.² This weakness, however, illustrates the general challenge of developing a new proof framework. As we can attest, without engaging in an effort to formally verify the framework, there are likely to be conditions that make the framework unsound.

Fichte, Hecher, and Roland [12] devised the MICE proof framework for model counting programs. Their proof rules are based on the algorithms commonly used by model counters. They developed a program that can generate proof traces from dec-DNNF graphs and a program to check adherence to their proof rules. This framework is not directly comparable to ours, since it only certifies the unweighted model count, but it has similar goals. Again, they provide only informal arguments regarding the soundness of their framework.

Both of these prior certification frameworks are strongly tied to the algorithms used by the knowledge compilers and model counters. Some of the conditions to be checked are relevant only to specific implementations. Our framework is very general and is based on a small set of proof rules. It builds on the highly developed concepts of clausal proof systems. These factors were important in enabling formal verification. In Section 12 and the supplement [3], we also compare the performance of our toolchain to these other two.

¹ Downloaded May 18, 2023 as <https://github.com/crillab/d4/tree/333370cc1e843dd0749c1efe88516e72b5239174>.

² Available at <https://github.com/marijnheule/drat-trim/releases/tag/v05.22.2023>.

3 Logical Foundations

Let X denote a set of Boolean variables, and let α be an *assignment* of truth values to some subset of the variables, where 0 denotes false and 1 denotes true, i.e., $\alpha: X' \rightarrow \{0, 1\}$ for some $X' \subseteq X$. We say the assignment is *total* when it assigns a value to every variable ($X' = X$), and that it is *partial* otherwise. The set of all possible total assignments over X is denoted \mathcal{U} .

For each variable $x \in X$, we define the *literals* x and \bar{x} , where \bar{x} is the negation of x . An assignment α can be viewed as a set of literals, where we write $\ell \in \alpha$ when $\ell = x$ and $\alpha(x) = 1$ or when $\ell = \bar{x}$ and $\alpha(x) = 0$. We write the negation of literal ℓ as $\bar{\ell}$. That is, $\bar{\bar{\ell}} = \ell$ when $\ell = x$ and $\bar{\bar{\ell}} = x$ when $\ell = \bar{x}$.

► **Definition 1** (Boolean Formulas). *The set of Boolean formulas is defined recursively. Each formula ϕ has an associated dependency set $\mathcal{D}(\phi) \subseteq X$, and a set of models $\mathcal{M}(\phi)$, consisting of total assignments that satisfy the formula:*

1. Boolean constants 0 and 1 are Boolean formulas, with $\mathcal{D}(0) = \mathcal{D}(1) = \emptyset$, with $\mathcal{M}(0) = \emptyset$, and with $\mathcal{M}(1) = \mathcal{U}$.
2. Variable x is a Boolean formula, with $\mathcal{D}(x) = \{x\}$ and $\mathcal{M}(x) = \{\alpha \in \mathcal{U} \mid \alpha(x) = 1\}$.
3. For formula ϕ , its negation, written $\neg\phi$ is a Boolean formula, with $\mathcal{D}(\neg\phi) = \mathcal{D}(\phi)$ and $\mathcal{M}(\neg\phi) = \mathcal{U} - \mathcal{M}(\phi)$.
4. For formulas $\phi_1, \phi_2, \dots, \phi_k$, their product $\phi = \bigwedge_{1 \leq i \leq k} \phi_i$ is a Boolean formula, with $\mathcal{D}(\phi) = \bigcup_{1 \leq i \leq k} \mathcal{D}(\phi_i)$ and $\mathcal{M}(\phi) = \bigcap_{1 \leq i \leq k} \mathcal{M}(\phi_i)$.
5. For formulas $\phi_1, \phi_2, \dots, \phi_k$, their sum $\phi = \bigvee_{1 \leq i \leq k} \phi_i$ is a Boolean formula, with $\mathcal{D}(\phi) = \bigcup_{1 \leq i \leq k} \mathcal{D}(\phi_i)$ and $\mathcal{M}(\phi) = \bigcup_{1 \leq i \leq k} \mathcal{M}(\phi_i)$.

We highlight some special classes of Boolean formulas. A formula is in *negation normal form* when negation is applied only to variables. A formula is in *conjunctive normal form* (CNF) when i) it is in negation normal form, and ii) sum is applied only to literals. A CNF formula can be represented as a set of *clauses*, each of which is a set of literals. Each clause represents the sum of the literals, and the formula is the product of its clauses. We use set notation to reference the clauses within a formula and the literals within a clause. A clause consisting of a single literal is referred to as a *unit* clause and the literal as a *unit* literal. This literal must be assigned value 1 by any satisfying assignment of the formula.

► **Definition 2** (Partitioned-Operation Formula). *A partitioned-operation formula satisfies the following for all product and sum operations:*

1. *The arguments to each product must have disjoint dependency sets. That is, operation $\bigwedge_{1 \leq i \leq k} \phi_i$ requires $\mathcal{D}(\phi_i) \cap \mathcal{D}(\phi_j) = \emptyset$ for $1 \leq i < j \leq k$.*
2. *The arguments to each sum must have disjoint models. That is, operation $\bigvee_{1 \leq i \leq k} \phi_i$ requires $\mathcal{M}(\phi_i) \cap \mathcal{M}(\phi_j) = \emptyset$ for $1 \leq i < j \leq k$.*

We let \wedge^p and \vee^p denote the product and sum operations in a partitioned-operation formula.

4 Ring Evaluation of a Boolean Formula

We propose a general framework for summarizing properties of Boolean formulas along the lines of algebraic model counting [19].

► **Definition 3** (Commutative Ring). *A commutative ring \mathcal{R} is an algebraic structure $\langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, with elements in the set \mathcal{A} and with commutative and associative operations $+$ (addition) and \times (multiplication), such that multiplication distributes over addition. $\mathbf{0}$ is the additive identity and $\mathbf{1}$ is the multiplicative identity. Every element $a \in \mathcal{A}$ has an additive inverse $-a$ such that $a + -a = \mathbf{0}$.*

We write $a - b$ as a shorthand for $a + -b$.

► **Definition 4** (Ring Evaluation Problem). *For commutative ring \mathcal{R} , a ring weight function associates a value $w(x) \in \mathcal{A}$ with every variable $x \in X$. We then define $w(\bar{x}) \doteq \mathbf{1} - w(x)$.*

For Boolean formula ϕ and ring weight function w , the ring evaluation problem computes

$$\mathbf{R}(\phi, w) = \sum_{\alpha \in \mathcal{M}(\phi)} \prod_{\ell \in \alpha} w(\ell) \quad (1)$$

In this equation, sum \sum is computed using addition operation $+$, and product \prod is computed using multiplication operation \times .

Many important properties of Boolean formulas can be expressed as ring evaluation problems. The (standard) *model counting* problem for formula ϕ requires determining $|\mathcal{M}(\phi)|$. It can be cast as a ring evaluation problem by having $+$ and \times be addition and multiplication over rational numbers and using weight function $w(x) = 1/2$ for every variable x . Ring evaluation of formula ϕ gives the *density* of the formula, i.e., the fraction of all possible total assignments that are models. For $n = |X|$, scaling the density by 2^n yields the number of models. This formulation avoids the need for a “smoothing” operation, in which redundant expressions are inserted into the formula [10].

The *weighted model counting* problem is also defined over rational numbers. Some formulations allow independently assigning weights $W(x)$ and $W(\bar{x})$ for each variable x and its complement, with the possibility that $W(x) + W(\bar{x}) \neq 1$. We can cast this as a ring evaluation problem by letting $r(x) = W(x) + W(\bar{x})$, performing ring evaluation with weight function $w(x) = W(x)/r(x)$ for each variable x , and computing the weighted count as $\mathbf{R}(\phi, w) \times \prod_{x \in X} r(x)$. Of course, this requires that $r(x) \neq 0$ for all $x \in X$.

The *function hashing problem* provides a test of inequivalence for Boolean formulas. That is, for $n = |X|$, let \mathcal{R} be a finite field with $|\mathcal{A}| = m$ such that $m \geq 2n$. For each $x \in X$, choose a value from \mathcal{A} at random for $w(x)$. Two formulas ϕ_1 and ϕ_2 will clearly have $\mathbf{R}(\phi_1, w) = \mathbf{R}(\phi_2, w)$ if they are logically equivalent, and if $\mathbf{R}(\phi_1, w) \neq \mathbf{R}(\phi_2, w)$, then they are clearly inequivalent. If they are not equivalent, then the probability that $\mathbf{R}(\phi_1, w) \neq \mathbf{R}(\phi_2, w)$ will be at least $(1 - \frac{1}{m})^n \geq (1 - \frac{1}{2n})^n > 1/2$. Function hashing can therefore be used as part of a randomized algorithm for equivalence testing [2]. For example, it can compare different runs on a single formula, either from different compilers or from a single compiler with different configuration parameters.

5 Partitioned-Operation Graphs (POGs)

Performing ring evaluation on an arbitrary Boolean formula could be intractable, but it is straightforward for a formula with partitioned operations:

► **Proposition 5.** *Ring evaluation with operations \neg , \wedge^P , and \vee^P satisfies the following for any weight function w :*

$$\begin{aligned} \mathbf{R}(\neg\phi, w) &= \mathbf{1} - \mathbf{R}(\phi, w) \\ \mathbf{R}\left(\bigwedge_{1 \leq i \leq k}^P \phi_i, w\right) &= \prod_{1 \leq i \leq k} \mathbf{R}(\phi_i, w) \\ \mathbf{R}\left(\bigvee_{1 \leq i \leq k}^P \phi_i, w\right) &= \sum_{1 \leq i \leq k} \mathbf{R}(\phi_i, w) \end{aligned}$$

As is described in Section 10, we have proved these three equations using Lean 4.

A *partitioned-operation graph* (POG) is a directed, acyclic graph with nodes N and edges $E \subseteq N \times N$. We denote nodes with boldface symbols, such as \mathbf{u} and \mathbf{v} . When $(\mathbf{u}, \mathbf{v}) \in E$, node \mathbf{v} is said to be a *child* of node \mathbf{u} . The in- and out-degrees of node \mathbf{u} are defined as

$\text{indegree}(\mathbf{u}) = |E \cap (N \times \{\mathbf{u}\})|$, and $\text{outdegree}(\mathbf{u}) = |E \cap (\{\mathbf{u}\} \times N)|$. Node \mathbf{u} is said to be *terminal* if $\text{outdegree}(\mathbf{u}) = 0$. A terminal node is labeled by a Boolean constant or variable. Node \mathbf{u} is said to be *nonterminal* if $\text{outdegree}(\mathbf{u}) > 0$. A nonterminal node is labeled by Boolean operation \wedge^P or \vee^P . A node can be labeled with operation \wedge^P or \vee^P only if it satisfies the partitioning restriction for that operation. Every POG has a designated *root node* \mathbf{r} . Each edge has a *polarity*, indicating whether (negative polarity) or not (positive polarity) the corresponding argument should be negated.

A POG represents a partitioned-operation formula with a sharing of common subformulas. Every node in the graph can be viewed as a partitioned-operation formula, and so we write $\phi_{\mathbf{u}}$ as the formula denoted by node \mathbf{u} . Each such formula has a set of models, and we write $\mathcal{M}(\mathbf{u})$ as a shorthand for $\mathcal{M}(\phi_{\mathbf{u}})$.

We can now define and compare two related representations:

- A det-DNNF graph can be viewed a POG with negation applied only to variables.
- A dec-DNNF graph is a det-DNNF graph with the further restriction that any sum node \mathbf{u} has exactly two children \mathbf{u}_1 and \mathbf{u}_0 , and for these there is a *decision variable* x such that any total assignment $\alpha \in \mathcal{M}(\mathbf{u}_b)$ has $\alpha(x) = b$, for $b \in \{0, 1\}$.

The generalizations encompassed by POGs have also been referred to as *deterministic decomposable circuits* (d-Ds) [23]. Our current proof generator only works for knowledge compilers generating dec-DNNF representations, but these generalizations allow for future extensions, while maintaining the ability to efficiently perform ring evaluation.

We define the *size* of POG P , written $|P|$, to be the the number of nonterminal nodes plus the number of edges from these nodes to their children. Ring evaluation of P can be performed with at most $|P|$ ring operations by traversing the graph from the terminal nodes up to the root, computing a value $\mathbf{R}(\mathbf{u}, w)$ for each node \mathbf{u} . The final result is then $\mathbf{R}(\mathbf{r}, w)$.

6 Clausal Proof Framework

We write (possibly subscripted) θ for formulas encoded as clauses, possibly with extension variables. We write (possibly subscripted) ϕ for formulas that use no extension variables.

A proof in our framework consists of a sequence of clause addition and deletion steps, with each step preserving the set of solutions to the original formula. The status of the proof at any step is represented as a set of *active* clauses θ , i.e., those that have been added but not yet deleted. Our framework is based on *extended* resolution [27], where proof steps can introduce new *extension variables* encoding Boolean formulas over input and prior extension variables. Let Z denote the set of extension variables occurring in formula θ . Starting with θ equal to input formula ϕ_I , the proof must maintain the invariant that $\phi_I \Leftrightarrow \exists Z \theta$.

Clauses can be added in two different ways. One is when they serve as the *defining clauses* for an extension variable. This form occurs only when defining \wedge^P and \vee^P operations, as is described in Section 7. Clauses can also be added or deleted based on *implication redundancy*. That is, when clause C satisfies $\theta \Rightarrow C$ for formula θ , then it can either be added to θ to create the formula $\theta \cup \{C\}$ or it can be deleted from $\theta \cup \{C\}$ to create θ .

We use *reverse unit propagation* (RUP) to certify implication redundancy when adding or deleting clauses [13, 29]. RUP is the core rule supported by standard proof checkers [15, 30] for propositional logic. It provides a simple and efficient way to check a sequence of applications of the resolution proof rule [25]. Let $C = \{\ell_1, \ell_2, \dots, \ell_p\}$ be a clause to be proved redundant with respect to formula θ . Let D_1, D_2, \dots, D_k be a sequence of supporting *antecedent* clauses, such that each D_i is in θ . A RUP step proves that $\bigwedge_{1 \leq i \leq k} D_i \Rightarrow C$ by showing that the combination of the antecedents plus the negation of C leads to a contradiction. The negation

of C is the formula $\bar{\ell}_1 \wedge \bar{\ell}_2 \wedge \dots \wedge \bar{\ell}_p$, having a CNF representation consisting of p unit clauses of the form $\bar{\ell}_i$ for $1 \leq i \leq p$. A RUP check processes the clauses of the antecedent in sequence, inferring additional unit clauses. In processing clause D_i , if all but one of the literals in the clause is the negation of one of the accumulated unit clauses, then we can add this literal to the accumulated set. That is, all but this literal have been falsified, and so it must be set to true for the clause to be satisfied. The final step with clause D_k must cause a contradiction, i.e., all of its literals are falsified by the accumulated unit clauses.

Compared to the proofs of unsatisfiability generated by SAT solvers, ours have important differences. Most significantly, each proof step must preserve the set of solutions with respect to the input variables; our proofs must therefore justify both clause deletions and additions. By contrast, an unsatisfiability proof need only guarantee that no proof step causes a satisfiable set of clauses to become unsatisfiable, and therefore it need only justify clause additions.

7 The CPOG Representation and Proof System

A CPOG file provides both a declaration of a POG, as well as a checkable proof that a Boolean formula, given in conjunctive normal form, is logically equivalent to the POG. The proof format draws its inspiration from the LRAT [14] and QRAT [16] formats for unquantified and quantified Boolean formulas, respectively. Key properties include:

- The file contains declarations of \wedge^P and \vee^P operations to describe the POG. Declaring a node u implicitly adds an *extension* variable u and a set of *defining* clauses θ_u encoding the product or sum operation. This is the only means for adding extension variables to the proof.
- Boolean negation is supported implicitly by allowing the arguments of the \vee^P and \wedge^P operations to be literals and not just variables.
- The file contains explicit clause addition steps. A clause can only be added if it is logically implied by the existing clauses. A sequence of clause identifiers must be listed as a *hint* providing a RUP verification of the implication.
- The file contains explicit clause deletion steps. A clause can only be deleted if it is logically implied by the remaining clauses. A sequence of clause identifiers must be listed as a *hint* providing a RUP verification of the implication.
- The checker must track the dependency set for every input and extension variable. For each \wedge^P operation, the checker must ensure that the dependency sets for its arguments are disjoint. The associated extension variable has a dependency set equal to the union of those of its arguments.
- Declaring a \vee^P operation requires a sequence of clauses providing a RUP proof that the arguments are mutually exclusive. Only binary \vee^P operations are allowed to avoid requiring multiple proofs of disjointness

7.1 Syntax

Table 1 shows the declarations that can occur in a CPOG file. As with other clausal proof formats, a variable is represented by a positive integer v , with the first ones being input variables and successive ones being extension variables. Literal ℓ is represented by a signed integer, with $-v$ being the logical negation of variable v . Each clause is indicated by a positive integer identifier C , with the first ones being the IDs of the input clauses and successive ones being the IDs of added clauses. Clause identifiers must be totally ordered, such that any clause identifier C' given in the hint when adding clause C must have $C' < C$.

■ **Table 1** CPOG Step Types. C : clause identifier, L : literal, V : variable.

		Rule		Description
C	a	$L^* 0$	$C^+ 0$	Add RUP clause
	d	C	$C^+ 0$	Delete RUP clause
C	p	$V L^* 0$		Declare \wedge^p operation
C	s	$V L L$	$C^+ 0$	Declare \vee^p operation
	r	L		Declare root literal

■ **Table 2** Defining Clauses for Product (A) and Sum (B) Operations.

(A). Product Operation \wedge^p						(B). Sum Operation \vee^p			
ID	Clause					ID	Clause		
i	v	$-\ell_1$	$-\ell_2$	\dots	$-\ell_k$	i	$-v$	ℓ_1	ℓ_2
$i+1$	$-v$	ℓ_1				$i+1$	v	$-\ell_1$	
$i+2$	$-v$	ℓ_2				$i+2$	v	$-\ell_2$	
		\dots							
$i+k$	$-v$	ℓ_k							

The first set of proof rules are similar to those in other clausal proofs. Clauses can be added via RUP addition (command **a**), with a sequence of antecedent clauses (the “hint”). Similarly for clause deletion (command **d**).

The declaration of a *product* operation, creating a node with operation \wedge^p , has the form:

$$i \quad \mathbf{p} \quad v \quad \ell_1 \quad \ell_2 \quad \dots \quad \ell_k \quad 0$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \dots, \ell_k$ is a sequence of k integers, indicating the arguments as literals of existing variables. As Table 2(A) shows, this declaration implicitly causes $k + 1$ clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the product of its arguments.

The dependency sets for the arguments represented by each pair of literals ℓ_i and ℓ_j must be disjoint, for $1 \leq i < j \leq k$. A product operation may have no arguments, representing Boolean constant 1. The only clause added to the proof will be the unit literal v . A reference to literal $-v$ then provides a way to represent constant 0.

The declaration of a *sum* operation, creating a node with operation \vee^p , has the form:

$$i \quad \mathbf{s} \quad v \quad \ell_1 \quad \ell_2 \quad H \quad 0$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and ℓ_1 and ℓ_2 are signed integers, indicating the arguments as literals of existing variables. Hint H consists of a sequence of clause IDs, all of which must be defining clauses for other POG operations.³ As Table 2(B) shows, this declaration implicitly causes three

³ The restriction to defining clauses in the hint is critical to soundness. Allowing the hint to include the IDs of input clauses creates an exploitable weakness. We discovered this weakness in the course of our efforts at formal verification.

clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the sum of its arguments. The hint must provide a RUP proof of the clause $\bar{\ell}_1 \vee \bar{\ell}_2$, showing that the two children of this node have disjoint models.

Finally, the literal denoting the root of the POG is declared with the `r` command. It can occur anywhere in the file. Except in degenerate cases, it will be the extension variable representing the root of a graph.

7.2 Semantics

The defining clauses for a product or sum operation uniquely define the value of its extension variable for any assignment of values to the argument variables. For the extension variable u associated with any POG node \mathbf{u} , we can therefore prove that any total assignment α to the input variables that also satisfies the POG defining clauses will assign a value to u such that $\alpha(u) = 1$ if and only if $\alpha \in \mathcal{M}(\mathbf{u})$.

The sequence of operator declarations, asserted clauses, and clause deletions represents a systematic transformation of the input formula into a POG. Validating all of these steps serves to prove that POG P is logically equivalent to the input formula. At the completion of the proof, the following FINAL CONDITIONS must hold:

1. There is exactly one remaining clause that was added via RUP addition, and this is a unit clause consisting of root literal r .
2. All of the input clauses have been deleted.

In other words, at the end of the proof it must hold that the active clauses be exactly those in $\theta_P \doteq \{\{r\}\} \cup \bigcup_{\mathbf{u} \in P} \theta_u$, the formula consisting of unit clause $\{r\}$ and the defining clauses for the nodes, providing a Tseitin encoding of P . Recognizing that any total assignment to the input variables implicitly defines the assignments to the extension variables, we can see that θ_P is the clausal encoding of ϕ_r . Let ϕ_I denote the input formula. The sequence of clause addition steps provides a *forward implication* proof that $\mathcal{M}(\phi_I) \subseteq \mathcal{M}(\phi_r)$. That is, any total assignment α satisfying the input formula must also satisfy the formula represented by the POG. Conversely, each proof step that deletes an input clause proves that any total assignment α that falsifies the clause must falsify ϕ_r . Deleting all but the final asserted clause and all input clauses provides a *reverse implication* proof that $\mathcal{M}(\phi_r) \subseteq \mathcal{M}(\phi_I)$.

The supplement [3], shows the CPOG description for an input formula with five clauses, yielding a POG with six nonterminal nodes. It explains how the clause addition and deletion steps yield a proof of equivalence between the input formula and its POG representation.

8 Generating CPOG from dec-DNNF

A dec-DNNF graph can be directly translated into a POG. In doing this conversion, our program performs simplifications to eliminate Boolean constants. Except in degenerate cases, where the formula is unsatisfiable or a tautology, we can therefore assume that the POG does not contain any constant nodes. In addition, negation is only applied to variables, and so the only edges with negative polarity will have variables as children. We can therefore view the POG as consisting of *literal* nodes corresponding to input variables and their negations, along with *nonterminal* nodes, which can be further classified as *product* and *sum* nodes.

8.1 Forward Implication Proof

For input formula ϕ_I and its translation into a POG P with root node \mathbf{r} , the most challenging part of the proof is to show that $\mathcal{M}(\phi_I) \subseteq \mathcal{M}(\phi_r)$, i.e., that any total assignment α that is a model of ϕ_I and the POG definition clauses will yield $\alpha(r) = 1$, for root literal r . This part of the proof consists of a series of clause assertions leading to one adding $\{r\}$ as a unit

clause. We have devised two methods of generating this proof. The *monolithic* approach makes just one call to a proof-generating SAT solver and has it determine the relationship between the two representations. The *structural* approach recursively traverses the POG, generating proof obligations at each node encountered. It may require multiple calls to a proof-generating SAT solver.

As notation, let ψ be a subset of the clauses in ϕ_I . For partial assignment ρ , the expression $\psi|_\rho$ denotes the set of clauses γ obtained from ψ by: i) eliminating any clause containing a literal ℓ such that $\rho(\ell) = 1$, ii) for the remaining clauses eliminating those literals ℓ for which $\rho(\ell) = 0$, and iii) eliminating any duplicate or tautological clauses. In doing these simplifications, we also track the *provenance* of each simplified clause C , i.e., which of the (possibly multiple) input clauses simplified to become C . More formally, for $C \in \psi|_\rho$, we let $\text{Prov}_\rho(C, \psi)$ denote those clauses $C' \in \psi$, such that $C' \subseteq C \cup \bigcup_{\ell \in \rho} \bar{\ell}$. We then extend the definition of Prov to any simplified formula γ as $\text{Prov}_\rho(\gamma, \psi) = \bigcup_{C \in \gamma} \text{Prov}_\rho(C, \psi)$.

The monolithic approach takes advantage of the clausal representations of the input formula ϕ_I and the POG formula ϕ_r . We can express the negation of ϕ_r in clausal form as $\theta_{\bar{r}} \doteq \bigcup_{u \in P} \theta_u|_{\{\bar{r}\}}$. Forward implication will hold when $\phi_I \Rightarrow \phi_r$, or equivalently when the formula $\phi_I \wedge \theta_{\bar{r}}$ is unsatisfiable, where the conjunction can be expressed as the union of the two sets of clauses. The proof generator writes the clauses to a file and invokes a proof-generating SAT solver. For each clause C in the unsatisfiability proof, it adds clause $\{r\} \cup C$ to the CPOG proof, and so the empty clause in the proof becomes the unit clause $\{r\}$. Our experimental results show that this approach can be very effective and generates short proofs for smaller problems, but it does not scale well enough for general use.

We describe the structural approach to proof generation as a recursive procedure $\text{validate}(\mathbf{u}, \rho, \psi)$ taking as arguments POG node \mathbf{u} , partial assignment ρ , and a set of clauses $\psi \subseteq \phi_I$. The procedure adds a number of clauses to the proof, culminating with the addition of the *target* clause: $u \vee \bigvee_{\ell \in \rho} \bar{\ell}$, indicating that $(\bigwedge_{\ell \in \rho} \ell) \Rightarrow u$, i.e., that any total assignment α such that $\rho \subseteq \alpha$ will assign $\alpha(u) = 1$. The top-level call has $\mathbf{u} = \mathbf{r}$, $\rho = \emptyset$, and $\psi = \phi_I$. The result will therefore be to add unit clause $\{r\}$ to the proof. Here we present a correct, but somewhat inefficient formulation of validate . We then refine it with some optimizations.

The recursive call $\text{validate}(\mathbf{u}, \rho, \psi)$ assumes that we have traversed a path from the root node down to node \mathbf{u} , with the literals encountered in the product nodes forming the partial assignment ρ . The set of clauses ψ can be a proper subset of the input clauses ϕ_I when a product node has caused a splitting into clauses containing disjoint variables. The subgraph with root node \mathbf{u} should be a POG representation of the formula $\psi|_\rho$.

The process for generating such a proof depends on the form of node \mathbf{u} :

1. If \mathbf{u} is a literal ℓ' , then the formula $\psi|_\rho$ must consist of the single unit clause $C = \{\ell'\}$, such that any $C' \in \text{Prov}_\rho(C, \psi)$ must have $C' \subseteq \{\ell'\} \cup \bigcup_{\ell \in \rho} \bar{\ell}$. Any of these can serve as the target clause.
2. If \mathbf{u} is a sum node with children \mathbf{u}_1 and \mathbf{u}_0 , then, since the node originated from a dec-DNNF graph, there must be some variable x such that either \mathbf{u}_1 is a literal node for x or \mathbf{u}_1 is a product node containing a literal node for x as a child. In either case, we recursively call $\text{validate}(\mathbf{u}_1, \rho \cup \{x\}, \psi)$. This will cause the addition of the target clause $u_1 \vee \bar{x} \vee \bigvee_{\ell \in \rho} \bar{\ell}$. Similarly, either \mathbf{u}_0 is a literal node for \bar{x} or \mathbf{u}_0 is a product node containing a literal node for \bar{x} as a child. In either case, we recursively call $\text{validate}(\mathbf{u}_0, \rho \cup \{\bar{x}\}, \psi)$, causing the addition of the target clause $u_0 \vee x \vee \bigvee_{\ell \in \rho} \bar{\ell}$. These recursive results can be combined with the second and third defining clauses for \mathbf{u} (see Table 2(B)) to generate the target clause for \mathbf{u} , requiring at most two RUP steps.
3. If \mathbf{u} is a product node, then we can divide its children into a set of literal nodes λ and a set of nonterminal nodes $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$.

- a. For each literal $\ell \in \lambda$, we must prove that any total assignment α , such that $\rho \subset \alpha$ has $\alpha(\ell) = 1$. In some cases, this can be done by simple Boolean constraint propagation (BCP). In other cases, we must prove that the formula $\psi|_{\rho \cup \{\bar{\ell}\}}$ is unsatisfiable. We do so by writing the formula to a file, invoking a proof-generating SAT solver, and then converting the generated unsatisfiability proof into a sequence of clause additions in the CPOG file. (The solver is constrained to only use RUP inference rules, preventing it from introducing extension variables.)
 - b. For a single nonterminal child ($k = 1$), we recursively call `validate` ($\mathbf{u}_1, \rho \cup \lambda, \psi$).
 - c. For multiple nonterminal children ($k > 1$), it must be the case that the clauses in $\gamma = \psi|_{\rho}$ can be partitioned into k subsets $\gamma_1, \gamma_2, \dots, \gamma_k$ such that $\mathcal{D}(\gamma_i) \cap \mathcal{D}(\gamma_j) = \emptyset$ for $1 \leq i < j \leq k$, and we can match each node \mathbf{u}_i to subset γ_i based on its literals. For each i such that $1 \leq i \leq k$, let $\psi_i = \text{Prov}_{\rho}(\gamma_i, \psi)$, i.e., those input clauses in ψ that, when simplified, became clause partition γ_i . We recursively call `validate` ($\mathbf{u}_i, \rho \cup \lambda, \psi_i$).
- We then generate the target clause for node \mathbf{u} with a single RUP step, creating the hint by combining the results from the BCP and SAT calls for the literals, the recursively computed target clauses, and all but the first defining clause for node \mathbf{u} (see Table 2(A)).

Observe that all of these steps involve a polynomial number of operations per recursive call, with the exception of those that call a SAT solver to validate a literal.

8.2 Reverse Implication Proof

Completing the equivalence proof of input formula ϕ_I and its POG representation with root node \mathbf{r} requires showing that $\mathcal{M}(\phi_{\mathbf{r}}) \subseteq \mathcal{M}(\phi_I)$. This is done in the CPOG framework by first deleting all asserted clauses, except for the final unit clause for root literal r , and then deleting all of the input clauses.

The asserted clauses can be deleted in reverse order, using the same hints that were used in their original assertions. By reversing the order, those clauses that were used in the hint when a clause was added will still remain when it is deleted.

Each input clause deletion can be done as a single RUP step, based on an algorithm to test for clausal entailment in det-DNNF graphs [4, 10]. The proof generator constructs the hint sequence from the defining clauses of the POG nodes via a single, bottom-up pass through the graph. The RUP deletion proof for input clause C effectively proves that any total assignment α that satisfies the POG definition clauses but does not satisfy C will yield $\alpha(r) = 0$. It starts with the set of literals $\{\bar{\ell} \mid \ell \in C\}$, describing the required condition for assignment α to falsify clause C . It then adds literals via unit propagation until a conflict arises. Unit literal r gets added right away, setting up a potential conflict.

Working upward through the graph, node \mathbf{u} is *marked* when the collected set of literals forces u to evaluate to 0. When marking \mathbf{u} , the program adds \bar{u} to the RUP literals and adds the appropriate defining clause to the hint. A literal node for ℓ will be marked if $\ell \in C$, with no hint required. If product node \mathbf{u} has some child \mathbf{u}_i that is marked, then \mathbf{u} is marked and clause $i + 1$ from among its defining clauses (see Table 2(A)) is added to the hint. Marking sum node \mathbf{u} requires that its two children are marked. The first defining clause for this node (see Table 2(B)) will then be added to the hint. At the very end, the program (assuming the reverse implication holds) will attempt to mark root node \mathbf{r} , which would require $\alpha(r) = 0$, yielding a conflict.

It can be seen that the reverse implication proof will be polynomial in the size of the POG, because each clause deletion requires a single RUP step having a hint with length bounded by the number of POG nodes.

9 Optimizations

The performance of the structural proof generator for forward implication, both in its execution time and the size of the proof generated, can be improved by two optimizations described here. A key feature is that they do not require any changes to the proof framework – they build on the power of extended resolution to enable new logical structures. They involve declaring new product nodes to encode products of literals. These nodes are not part of the POG representation of the formula; they serve only to enable the forward implication proof. Here we summarize the two optimizations. The supplement [3] provides more details.

Literal Grouping: A single recursive step of `validate` can encounter product nodes having many – tens or even hundreds – of literals as children. The earlier formulation of `validate` considers each literal $\ell \in \lambda$ separately, calling a SAT solver for every literal that cannot be validated with BCP. Literal grouping handles all of these literals together. It defines product node \mathbf{v} having the literals as children. The goal then becomes to prove that any total assignment must yield 1 for extension variable v . Calling a solver with v set to 0 yields an unsatisfiability proof that can be mapped back to a sequence of clause additions in the CPOG file validating all of the literals.

Lemmas: Our formulation of `validate` requires each call at a node \mathbf{u} to recursively validate all of its children. This effectively expands the graph into a tree, potentially requiring an exponential number of recursive steps. Instead, for each node \mathbf{u} having $\text{indegree}(\mathbf{u}) > 1$, the program can define and generate the proof of a *lemma* for \mathbf{u} when it is first reached by a call to `validate` and then apply this lemma for this and subsequent calls. The lemma states that the POG with root node \mathbf{u} satisfies forward implication for a formula $\gamma_{\mathbf{u}}$, where some of the clauses in this formula are input clauses from ϕ_I , but others are simplified versions of input clauses. The key idea is to introduce product nodes to encode (via DeMorgan’s Laws) the simplified clauses and have these serve as lemma arguments.

The combination of these two optimization guarantees that i) each call to `validate` for a product node will cause at most one invocation of the SAT solver, and ii) each call to `validate` for any node \mathbf{u} will cause further recursive calls only once. Our experimental results [3] show that these optimizations yield substantial benefits.

10 A Formally Verified Toolchain

We set out to formally verify the system with two goals in mind: first, to ensure that the CPOG framework is mathematically sound; and second, to implement correct-by-construction proof checking and ring evaluation (the “Trusted Code” components of Figure 1). These two goals are achieved with a single proof development in the Lean 4 programming language [11]. Verification was greatly aided by the Aesop [22] automated tactic. In this section, we briefly describe the functionality we implemented and what we proved about it. More information is provided in the supplement [3].

Proof checking. The goal of a CPOG proof is to construct a POG that is equivalent to the input CNF ϕ_I . The POG being constructed, and the set of active clauses are stored in the checker state `st`. The checker begins by parsing the input formula, initializing the active clauses to $\theta \leftarrow \phi_I$, and initializing the POG P to an empty one. It then processes every step of the CPOG proof, either modifying its state by adding/deleting clauses in θ and adding nodes to P , or throwing an exception if a step is incorrect. Afterwards, it carries out the FINAL CONDITIONS check of Section 7.2. Throughout the process, we maintain invariants that ensure that P is partitioned and that a successful final check entails the logical equivalence of ϕ_I and $\phi_{\mathbf{r}}$, where \mathbf{r} is the final POG root (Theorem 6).

6:14 Certified Knowledge Compilation

The specifications we use to state these invariants are built on a general theory of propositional logic, mirroring Section 3. Following the DIMACS CNF convention, we define the data types `Var` of variables being positive natural numbers, `ILit` of literals being non-zero integers, and `PropForm` of propositional formulas. `PropForm` is generic over the type of variables, so we instantiate it with our type as `PropForm Var`. Assignments of truth values are taken to be total functions `PropAssignment Var := Var → Bool`. Requiring totality is not a limitation: instead of talking about two equal, partial assignments to a subset $X' \subseteq X$ of variables, we can more conveniently talk about two total assignments that agree on X' . We write $\sigma \models \varphi$ when $\sigma : \text{PropAssignment Var}$ satisfies $\varphi : \text{PropForm Var}$.

The invariants refer to the checker state `st` with fields `st.inputCnf` for ϕ_I , `st.clauseDb` for θ , `st.pog` for P , `st.pogDefsForm` for the clausal POG definitions formula $\bigwedge_{u \in P} \theta_u$, and `st.allVars` for all variables (original and extension) added so far. For any $\mathbf{u} \in P$, `st.pog.toPropForm u` computes $\phi_{\mathbf{u}}$. The first two invariants state that assignments to original variables extend uniquely to extension variables defining the POG nodes. In the formalization, we split this into extension and uniqueness:

```

/-- Any assignment satisfying  $\varphi_1$  extends to  $\varphi_2$  while preserving values on X. -/
def extendsOver (X : Set Var) ( $\varphi_1 \varphi_2 : \text{PropForm Var}$ ) :=
   $\forall (\sigma_1 : \text{PropAssignment Var}), \sigma_1 \models \varphi_1 \rightarrow \exists \sigma_2, \sigma_1.\text{agreeOn } X \sigma_2 \wedge \sigma_2 \models \varphi_2$ 
/-- Assignments satisfying  $\varphi$  are determined on Y by their values on X. -/
def uniqueExt (X Y : Set Var) ( $\varphi : \text{PropForm Var}$ ) :=
   $\forall (\sigma_1 \sigma_2 : \text{PropAssignment Var}), \sigma_1 \models \varphi \rightarrow \sigma_2 \models \varphi \rightarrow \sigma_1.\text{agreeOn } X \sigma_2 \rightarrow$ 
     $\sigma_1.\text{agreeOn } Y \sigma_2$ 

```

```

invariants.extends_pogDefsForm : extendsOver st.inputCnf.vars  $\top$  st.pogDefsForm
invariants.uep_pogDefsForm : uniqueExt st.inputCnf.vars st.allVars st.pogDefsForm

```

Note that in the definition of `uniqueExt`, the arrows associate to the right, so the definition says that the three assumptions imply the conclusion. The next invariant guarantees that the set of solutions over the original variables is preserved:

```

def equivalentOver (X : Set Var) ( $\varphi_1 \varphi_2 : \text{PropForm Var}$ ) :=
  extendsOver X  $\varphi_1 \varphi_2 \wedge$  extendsOver X  $\varphi_2 \varphi_1$ 

```

```

invariants.equivInput : equivalentOver st.inputCnf.vars st.inputCnf st.clauseDb

```

Finally, for every node $\mathbf{u} \in P$ with corresponding literal u we ensure that $\phi_{\mathbf{u}}$ is partitioned (Definition 2) and relate $\phi_{\mathbf{u}}$ to its clausal encoding $\theta_u \doteq u \wedge \bigwedge_{v \in P} \theta_v$:

```

def partitioned : PropForm Var → Prop
  | tr | fls | var _ => True
  | neg  $\varphi$  =>  $\varphi.\text{partitioned}$ 
  | disj  $\varphi \psi$  =>  $\varphi.\text{partitioned} \wedge \psi.\text{partitioned} \wedge \forall \tau, \neg(\tau \models \varphi \wedge \tau \models \psi)$ 
  | conj  $\varphi \psi$  =>  $\varphi.\text{partitioned} \wedge \psi.\text{partitioned} \wedge \varphi.\text{vars} \cap \psi.\text{vars} = \emptyset$ 

invariants.partitioned :  $\forall (u : \text{ILit}), (\text{st.pog.toPropForm } u).\text{partitioned}$ 
invariants.equivalent_lits :  $\forall (u : \text{ILit}), \text{equivalentOver st.inputCnf.vars}$ 
   $(u \wedge \text{st.pogDefsForm}) (\text{st.pog.toPropForm } x)$ 

```

These invariants are maintained by valid CPOG proofs. Together with additional invariants that ensure the correctness of cached computations, they imply the soundness theorem for P with root node `r`:

► **Theorem 6.** *If the proof checker has assembled POG P with root node \mathbf{r} starting from input formula ϕ_I , and FINAL CONDITIONS (as stated in Section 7.2) hold of the checker state, then ϕ_I is logically equivalent to $\phi_{\mathbf{r}}$.*

Proof. FINAL CONDITIONS imply that the active clausal formula θ is exactly $\theta_P \doteq \{\{r\}\} \cup \bigcup_{u \in P} \theta_u$. The conclusion follows from this and the checker invariants. The full proof is formally verified in Lean. ◀

After certifying a CPOG proof, the checker can pass its in-memory POG representation to the ring evaluator, along with the partitioning guarantee provided by `invariants.partitioned`.

Ring evaluation. We formalized the central quantity (1) in the ring evaluation problem (Definition 4) in a commutative ring R as follows:

```
def weightSum {R : Type} [CommRing R]
  (weight : Var → R) (φ : PropForm Var) (s : Finset Var) : R :=
  Σ τ in models φ s, ∏ x in s, if τ x then weight x else 1 - weight x
```

The rules for efficient ring evaluation of partitioned formulas are expressed as:

```
def ringEval (weight : Var → R) : PropForm Var → R
| tr      => 1
| fls     => 0
| var x   => weight x
| neg φ   => 1 - ringEval weight φ
| disj φ ψ => ringEval weight φ + ringEval weight ψ
| conj φ ψ => ringEval weight φ * ringEval weight ψ
```

Proposition 5 is then formalized as follows:

```
theorem ringEval_eq_weightSum (weight : Var → R) {φ : PropForm Var} :
  partitioned φ → ringEval weight φ = weightSum weight φ (vars φ)
```

To efficiently compute the ring evaluation of a formula represented by a POG node, we implemented `Pog.ringEval` and then proved that it matches the specification above:

```
theorem ringEval_eq_ringEval (pog : Pog) (weight : Var → R) (x : Var) :
  pog.ringEval weight x = (pog.toPropForm x).ringEval weight
```

Applying this to the output of our verified CPOG proof checker, which we know to be partitioned and equivalent to the input formula ϕ_I , we obtain a proof that our toolchain computes the correct ring evaluation of ϕ_I .

Model counting. Finally, we established that ring evaluation with the appropriate weights corresponds to the standard model count. To do so, we defined a function that carries out an integer calculation of the number of models over a set of variables of cardinality `nVars`:

```
def countModels (nVars : Nat) : PropForm Var → Nat
| tr      => 2^nVars
| fls     => 0
| var _   => 2^(nVars - 1)
| neg φ   => 2^nVars - countModels nVars φ
| disj φ ψ => countModels nVars φ + countModels nVars ψ
| conj φ ψ => countModels nVars φ * countModels nVars ψ / 2^nVars
```


We then formally proved that for a partitioned formula whose variables are among a finite set s , this computation really does count the number of models over s :

```
theorem countModels_eq_card_models { $\varphi$  : PropForm Var} { $s$  : Finset Var} :
  vars  $\varphi \subseteq s \rightarrow$  partitioned  $\varphi \rightarrow$  countModels (card  $s$ )  $\varphi =$  card (models  $\varphi s$ )
```

In particular, taking s to be exactly the variables of φ , we have that the number of models of φ on its variables is `countModels φ (card (vars φ))`.

11 Implementations

We have implemented programs that, along with the D4 knowledge compiler, form the toolchain illustrated in Figure 1.⁴ The proof generator is the same in both cases, since it need not be trusted. Our *verified* version of the proof checker and ring evaluator have been formally verified within the Lean 4 theorem prover. Our long term goal is to rely on these. Our *prototype* version is written in C. It is faster and more scalable, but we anticipate its need will diminish as the verified version is further optimized.

Our proof generator is written in C/C++ and uses CADICAL [1] as its SAT solver. To convert proof steps back into hinted CPOG clause additions, the generator can use either its own RUP proof generator, or it can invoke DRAT-TRIM [15]. The latter yields shorter proofs and scales well to large proofs, but each invocation has a high startup cost. We therefore only use it when solving larger problems (currently ones with over 1000 clauses).

The proof generator can optionally be instructed to generate a *one-sided* proof, providing only the reverse-implication portion of the proof via input clause deletion. This can provide useful information – any assignment that is a model for the compiled representation must also be a model for the input formula – even when full validation is impractical.

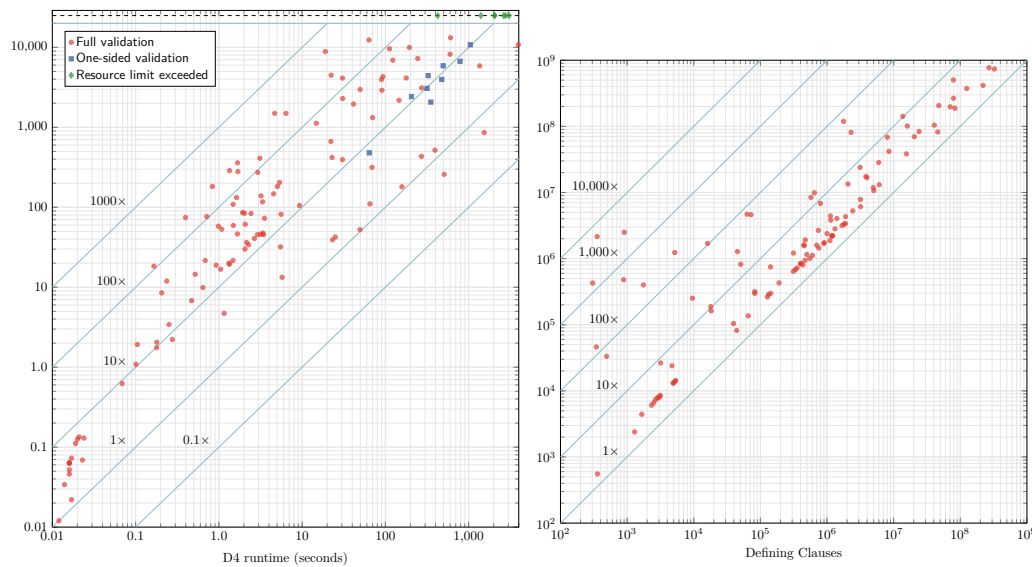
We incorporated a ring evaluator into the prototype checker. It can perform both standard and weighted model counting with full precision. It performs arithmetic over a subset of the rationals we call $\mathbb{Q}_{2,5}$, consisting of numbers of the form $a \cdot 2^b \cdot 5^c$, for integers a , b , and c , and with a implemented to have arbitrary range. Allowing scaling by powers of 2 enables the density computation and rescaling required for standard model counting. Allowing scaling by powers of both 2 and 5 enables exact decimal arithmetic, handling the weights used in the weighted model counting competitions. To give a sense of scale, the counter generated a result with 260,909 decimal digits for one of the weighted benchmarks.

12 Experimental Evaluation

We summarize the results of our experiments here. The supplement [3] provides a more complete description. For our evaluation, we used the public benchmark problems from the 2022 standard and weighted model competitions.⁵ We found that there were 180 unique CNF files among these, ranging in size from 250 to 2,753,207 clauses. We ran our programs on a processor with 64 GB of memory and having an attached high-speed, solid-state disk. With a runtime limit of 4000 seconds, D4 completed for 124 of the benchmark problems. Our proof generator was able to convert all of these into POGs, with their declarations ranging from 304 to 2,761,457,765 (median 848,784) defining clauses.

⁴ The source code for all tools is available at <https://github.com/rebryant/cpog/releases/tag/v1.0.0>.

⁵ Downloaded from https://mccompetition.org/2022/mc_description.html



■ **Figure 2** Runtime (left) and proof size (right) for CPOG proofs. The runtime includes proof generation, checking, and model counting relative to the runtime for D4. The proof size is measured as total clauses relative to the number of defining clauses.

We ran our proof generator with a time limit of 10,000 seconds. The results are shown in Figure 2. The left-hand plot shows the elapsed time for the combination of proof generation, checking, and counting versus the time for D4. The proof generator was able to generate full proofs for 108 of the problems and one-sided proofs for an additional 9 of them, leaving just 7 with no verification. The prototype checker successfully verified all of the generated proofs. The longest runtime for the combination of proof generation, checking, and counting for a full proof was 13,145 seconds. Overall the ratio between the combined time for generation, checking, and counting versus the time for D4 had a harmonic mean of 5.5. The right-hand plot shows the total number of clauses in the CPOG file versus the number of defining clauses for the problems having full proofs. The ratio between the total number of clauses and the number of defining clauses had a harmonic mean of 3.13. To date, we have not found any errors in the dec-DNNF graphs generated by D4.

We found that the monolithic approach for generating the forward implication proof works well for smaller POGs (up to one million defining clauses), but it becomes inefficient for larger ones. These experiments suggest a possible hybrid approach, stopping the recursion of the structural approach and shifting to monolithic mode once the subgraph size is below some threshold. By using monolithic mode, we were also able to perform end-to-end verification of all but one of the benchmarks that could be verified without preprocessing, with a total time limit (including preprocessing) of 1,000 seconds.

We also found that our two optimizations: literal grouping and lemmas can provide substantial improvements in proof size and runtime. In the extreme cases, a lack of lemmas caused one proof to grow by a factor of 52.5, while a lack of literal grouping caused another proof to grow by a factor of 39.6. Overall, it is clear that these two optimizations are worthwhile, and sometimes critical, to success for some benchmarks.

Running the verified proof checker in Lean 4 required, on average (harmonic mean), around 5.9 times longer than the prototype checker. Encouragingly, the scaling trends were identical for the two solvers, indicating that the two checkers have similar asymptotic performance. We consider a factor of 5.9 to be acceptable for the assurance provided by formal verification.

In comparing other proof frameworks, we found that the CD4 toolchain ran very fast and could handle very large benchmarks. Even with a total time limit of 1,000 seconds, including the time for knowledge compilation, the CD4 toolchain completed 106 benchmarks, while the CPOG toolchain completed just 82. We found that the runtimes for the MICE toolchain versus our CPOG toolchain showed little correlation, reflecting the fact that the two solve different problems and use different approaches. In general, our CPOG toolchain showed better scaling, in part due to its ability to control the recursion through lemmas. Neither of these prior toolchains could perform end-to-end verification when the knowledge compilation was preceded by preprocessing.

13 Conclusions

This paper demonstrates a method for certifying the equivalence of two different representations of a Boolean formula: an input formula represented in conjunctive normal form, and a compiled representation that can then be used to extract useful information about the formula, including its weighted and unweighted model counts. It builds on the extensive techniques that have been developed for clausal proof systems, including extended resolution and reverse unit propagation, as well as established tools, such as proof-generating SAT solvers and DRAT-TRIM.

We are hopeful that having checkable proofs for knowledge compilers will allow them to be used in applications where high levels of trust are required, and that it will provide a useful tool for developers of knowledge compilers. Our experiments demonstrate that our toolchain can already handle problems nearly at the limits of current knowledge compilers. Further engineering and optimization of our proof generator and checker could improve their performance and capacity substantially. We are hopeful that our tool can be adapted to handle other knowledge compiler representations, such as sentential decision diagrams (SDDs) [9].

References

- 1 Armin Biere. CaDiCaL at the SAT Race 2019. In *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.
- 2 Manuel Blum, Ashok K. Chandra, and Mark N. Wegman. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10(2):80–82, 18 March 1980.
- 3 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Supplement to certified knowledge compilation with application to verified model counting, May 2023. doi:10.5281/zenodo.7966174.
- 4 Florent Capelli. Knowledge compilation languages as proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *LNCS*, pages 90–91, 2019.
- 5 Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-DNNF compilers. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

- 6 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Conference on Automated Deduction (CADE)*, volume 10395 of *LNCS*, pages 220–236, 2017.
- 7 Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2002.
- 8 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *European Conference on Artificial Intelligence*, pages 328–332, 2004.
- 9 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *International Joint Conference on Artificial Intelligence*, pages 819–826, 2011.
- 10 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 2002.
- 11 Leonardo de Moura and Sebastian Ulrich. The Lean 4 theorem prover and programming language. In *Conference on Automated Deduction (CADE)*, volume 12699 of *LNAI*, pages 625–635, 2021.
- 12 Johannes K Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- 13 Evgueni I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE)*, pages 886–891, 2003.
- 14 Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Nathan D. Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving*, volume 10499 of *LNCS*, pages 269–284, 2017.
- 15 Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan D. Wetzler. Verifying refutations with extended resolution. In *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 345–359, 2013.
- 16 Marijn J. H. Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 8562 of *LNCS*, pages 91–106, 2014.
- 17 Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 22:191–219, 2007.
- 18 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *LNCS*, pages 355–370, 2012.
- 19 Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, July 2017.
- 20 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *International Joint Conference on Artificial Intelligence*, pages 667–673, 2017.
- 21 Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi:10.1007/s10817-019-09525-z.
- 22 Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for Lean. In *Certified Programs and Proofs (CPP)*, pages 253–266. ACM, 2023. doi:10.1145/3573105.3575671.
- 23 Mikaël Monet and Dan Olteanu. Towards deterministic decomposable circuits for safe queries. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, 2018.
- 24 Umut Oztok and Adnan Darwiche. On compiling CNF into decision-DNNF. In *Constraint Programming (CP)*, volume 8656 of *LNCS*, pages 42–57, 2014.
- 25 John A. Robinson. A machine-oriented logic based on the resolution principle. *J.ACM*, 12(1):23–41, January 1965.
- 26 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*, volume 12652 of *LNCS*, pages 223–241, 2021.

- 27 G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983.
- 28 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.
- 29 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proc. of the 10th Int. Symposium on Artificial Intelligence and Mathematics (ISAIM 2008)*, 2008.
- 30 Nathan D. Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *LNCS*, pages 422–429, 2014.
- 31 Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE)*, pages 880–885, 2003.