Certified Knowledge Compilation with Application to Verified Model Counting

Randal E. Bryant
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA
Wojciech Nawrocki
Department of Philosophy, Carnegie Mellon University
Jeremy Avigad
Department of Philosophy, Carnegie Mellon University
Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University

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 deterministic, decomposable, negation normal form (d-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 generalization 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 the d-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 d-DNNF 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...

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* 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

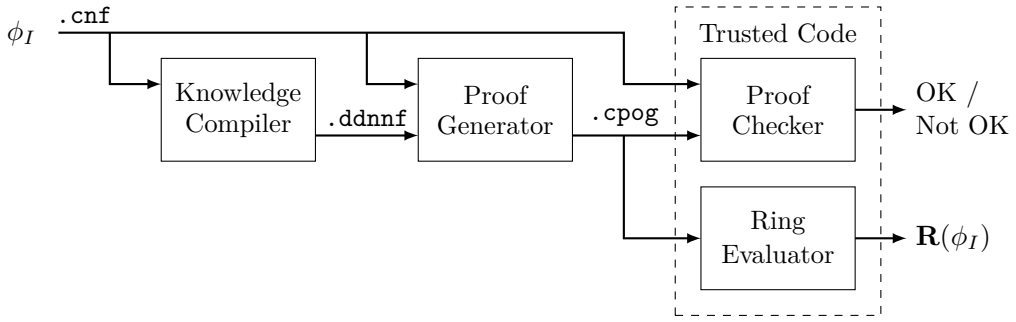


Figure 1 Certifying toolchain. The output of a standard knowledge compiler is converted into a combined graph/proof (CPOG) which can be independently checked and evaluated

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 [21]. 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* (d-DNNF) introduced by Darwiche [4, 5] represents 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 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* [13] framework describing properties of Boolean functions that can be efficiently computed from a d-DNNF representation. These include standard and weighted model counting, and much more.

One shortcoming of existing knowledge compilers is that they have no way to validate that their generated results are correct, i.e., 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 proof production capabilites 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 generalization 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 [19]. We establish a set of conditions that, when satisified by a CPOG file, guarantee 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 [14] generates a d-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 [7]. Running these two programs on a particular CPOG file gives assurance that the proof and the model count are correct. Our 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, subtle details in the partitioned sum rule were identified during this process.

We evaluate our toolchain using benchmark formulas from the 2022 standard and weighted model competitions. Our tools handle all but the largest d-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.

2 Related Work

Generating proofs of unsatisfiability in SAT solvers has a long tradition [24] and has become widely accepted due to the formulation of clausal proof systems for which proofs can readily be generated and efficiently checked [11, 23]. A number of formally verified checkers have been developed within different verification frameworks [3, 10, 15, 18]. The associated proofs add clauses while preserving satisfiability until the empty clause is derived. By contrast, 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.

Fichte, Hecher, and Roland [8] devised the MICE proof framework for model counting programs. Their proof rules are based on the algorithms commonly used by model counters. They present a series of lemmas and a theorem stating that their checking framework is sound. They also present experimental results for cases where they modify existing model counters to generate proof traces, and where they generate proof traces from d-DNNF graphs. Our toolchain is not directly comparable to theirs—they directly certify the numeric value produced by a model counter, while we certify that the graph generated by a knowledge compiler is logically equivalent to the input formula. We can then validate a standard or weighted count from that representation. Nonetheless, we compare the performance of our toolchain to theirs as part of our experimental evaluation in Appendix D.

We know of no other work on certifying the output of a knowledge compiler.

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., $\alpha\colon X' \to \{0,1\}$ 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 $\overline{x}$, where $\overline{x}$ 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 $\ell = \overline{x}$ and $\alpha(x) = 0$. We write the negation of literal ℓ as $\overline{\ell}$. That is, $\overline{\ell} = \overline{x}$ when $\ell = x$ and $\overline{\ell} = x$ when $\ell = \overline{x}$.

▶ **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 $\mathcal{M}(x) = \{\alpha \in \mathcal{U} | \alpha(x) = 1\}$.*
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 $\phi_1, \phi_2, \ldots, \phi_k$, 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 $\phi_1, \phi_2, \ldots, \phi_k$, 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 two 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 $\wedge^{\mathsf{p}}$ and $\vee^{\mathsf{p}}$ 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 [13].

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

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 $w(\overline{x}) \doteq \boldsymbol{1} - w(x)$.*

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

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

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 [6].

The *weighted model counting* problem is also defined over rational numbers. Some formulations allow independently assigning weights $W(x)$ and $W(\overline{x})$ for each variable x and its complement, with the possibility that $W(x) + W(\overline{x}) \neq 1$. We can cast this as a ring evaluation problem by letting $r(x) = W(x) + W(\overline{x})$, 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 $x \in X$, chose 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. If they are not equivalent, then the probability that $\mathbf{R}(\phi_1, w) \neq \mathbf{R}(\phi_2, w)$ will be at least $\left(1 - \frac{1}{m}\right)^n \geq \left(1 - \frac{1}{2n}\right)^n > 1/2$. 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 $\neg$, $\wedge^{\mathsf{P}}$, and $\vee^{\mathsf{P}}$ satisfies the following for any weight function w:*

$$\boldsymbol{R}(\neg\phi,\ w) = \boldsymbol{1} - \boldsymbol{R}(\phi, w) \tag{2}$$

$$\boldsymbol{R}\left(\bigwedge^{\mathsf{P}}{}_{1 \leq i \leq k}\, \phi_i,\ w\right) = \prod_{1 \leq i \leq k} \boldsymbol{R}(\phi_i, w) \tag{3}$$

$$\boldsymbol{R}\left(\bigvee^{\mathsf{P}}{}_{1 \leq i \leq k}\, \phi_i,\ w\right) = \sum_{1 \leq i \leq k} \boldsymbol{R}(\phi_i, w) \tag{4}$$

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 one of the three Boolean operations: $\neg$, $\wedge^{\mathsf{P}}$ or $\vee^{\mathsf{P}}$. Node $\mathbf{u}$ can be labeled with $\neg$ only if $\text{outdegree}(\mathbf{u}) = 1$. It can be labeled with operation $\wedge^{\mathsf{P}}$ or $\vee^{\mathsf{P}}$ only if it satisfies the partitioning restriction for that operation. Every POG has a designated *root node* $\mathbf{r}$.

In essence, a POG represents a partitioned-operation formula with a sharing of common subformulas. Indeed, 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 define the *size* of POG P, written $|P|$, to be the the number of nodes labeled $\wedge^{\mathsf{p}}$ or $\vee^{\mathsf{p}}$ 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 $\mathbf{R}(\mathbf{r}, w)$.

POGs generalize the d-DNNF graphs in two ways:

- They allow negation of arbitrary nodes in the graph, not just variables.
- They allow arbitrary arguments to a $\vee^{\mathsf{p}}$ operation, as long as these have disjoint models. By contrast, each sum node $\mathbf{u}$ in a d-DNNF must have two children $\mathbf{u}_1$ and $\mathbf{u}_0$, and for these there must be a *decision variable* x such that any total assignment $\alpha \in \mathcal{M}(\mathbf{u}_b)$ has $\alpha(x) = b$, for $b \in \{0, 1\}$.

These generalizations allow more flexibility in the POG representation while maintaining the ability to efficiently perform ring evaluation.

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 [19], 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 occuring 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 $\wedge^{\mathsf{p}}$ and $\vee^{\mathsf{p}}$ 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 [9, 22]. RUP is the core rule supported by standard proof checkers [11, 23] for propositional logic. It provides a simple and efficient way to check a sequence of applications of the resolution proof rule [17]. Let $C = \{\ell_1, \ell_2, \ldots, \ell_p\}$ be a clause to be proved redundant with respect to formula θ. Let $D_1, D_2, \ldots, D_k$ be a sequence of supporting *antecedent* clauses, such that each D_i is in θ. A RUP step proves that $\bigwedge_{1 \le i \le k} D_i \Rightarrow C$ by showing that the combination of the antecedents plus the negation of C leads to a contradiction. The negation of C is the formula $\overline{\ell}_1 \wedge \overline{\ell}_2 \wedge \cdots \wedge \overline{\ell}_p$, having a CNF representation consisting of p unit clauses of the form $\overline{\ell}_i$ for $1 \le i \le p$. 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

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 $\wedge^{\mathsf{p}}$ operation
C	s	V L L	C^+ 0	Declare $\vee^{\mathsf{p}}$ operation
	r	L		Declare root literal

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 POG 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 [10] and QRAT [12] formats for unquantified and quantified Boolean formulas, respectively. Key properties include:

- The file contains declarations of $\wedge^{\mathsf{p}}$ and $\vee^{\mathsf{p}}$ 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.
- Boolean negation is supported implicitly by allowing the arguments of the $\vee^{\mathsf{p}}$ and $\wedge^{\mathsf{p}}$ 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 $\wedge^{\mathsf{p}}$ 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 $\vee^{\mathsf{p}}$ operation requires a sequence of clauses providing a RUP proof that the arguments are mutually exclusive. Only binary $\vee^{\mathsf{p}}$ 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

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

(A). Product Operation $\wedge^{\mathtt{p}}$

ID	Clause				
i	v	$-\ell_1$	$-\ell_2$	$\cdots$	$-\ell_k$
$i{+}1$	$-v$	ℓ_1			
$i{+}2$	$-v$	ℓ_2			
	$\cdots$				
$i{+}k$	$-v$	ℓ_k			

(B). Sum Operation $\vee^{\mathtt{p}}$

ID	Clause		
i	$-v$	ℓ_1	ℓ_2
$i{+}1$	v	$-\ell_1$	
$i{+}2$	v	$-\ell_2$	

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 $C' < C$.

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 $\wedge^{\mathtt{p}}$, has the form:

$$i \quad \mathtt{p} \quad v \quad \ell_1 \quad \ell_2 \quad \cdots \quad \ell_k \quad \mathtt{0}$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \ldots, \ell_k$ is a sequence of k integers representing literals of existing variables. As Table 2(A) shows, this declaration implicitly causes $k+1$ clauses to be added to the proof, defining extension variable v to be 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 of representing constant 0.

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

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

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 representing literals of existing variables. Hint H consists of a sequence of clause IDs, all of which must be defining clauses for other POG operations. As Table 2(B) shows, this declaration implicitly causes three clauses to be added to the proof, defining extension variable v to be the sum of its arguments. The hint must provide a RUP proof of the clause $\overline{\ell}_1 \vee \overline{\ell}_2$, 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 which also satisfies the POG definitions 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 formula. Validating all of these steps serves to prove that the POG 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 of the POG nodes. 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 $\phi_{\mathbf{r}}$. Let ϕ_I denote the input formula. The sequence of clause addition steps provides a *forward implication* proof that $\mathcal{M}(\phi_I) \subseteq \mathcal{M}(\phi_{\mathbf{r}})$. 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 $\phi_{\mathbf{r}}$. Deleting all but the final asserted clause and all input clauses provides a *reverse implication* proof that $\mathcal{M}(\phi_{\mathbf{r}}) \subseteq \mathcal{M}(\phi_I)$.

Appendix A shows the CPOG description for an input formula with five clauses, yielding a POG with five 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 d-DNNF

A d-DNNF graph can be directly translated into a POG, although in doing this conversion, our program also performs simplifications to eliminate Boolean constants. Except in degenerate cases, we can therefore assume that the POG does not contain any constant nodes. In addition, negation is only applied to variables, and so we can merge the negation operations into the terminal nodes, viewing 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 $\mathcal{M}(\phi_I) \subseteq \mathcal{M}(\phi_{\mathbf{r}})$, 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 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 $\mathrm{Prov}_\rho(C, \psi)$ denote those

clauses $C' \in \psi$, such that $C' \subseteq C \cup \bigcup_{\ell \in \rho} \overline{\ell}$. 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 $\phi_{\mathbf{r}}$. We can express the negation of $\phi_{\mathbf{r}}$ in clausal form as $\theta_{\overline{\mathbf{r}}} \doteq \bigcup_{\mathbf{u} \in P} \theta_u|_{\{\overline{r}\}}$. Forward implication will hold when $\phi_I \Rightarrow \phi_{\mathbf{r}}$, or equivalently when the formula $\phi_I \wedge \theta_{\overline{\mathbf{r}}}$ 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 routine use.

We describe the structural approach to proof generation as a recursive procedure $\mathsf{validate}(\mathbf{u}, \rho, \psi)$ 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: $u \vee \bigvee_{\ell \in \rho} \overline{\ell}$, 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 $\mathsf{validate}$. We then refine it with some optimizations.

The recursive call $\mathsf{validate}(u, \rho, \psi)$ 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 $C' \subseteq \{\ell'\} \cup \bigcup_{\ell \in \rho} \overline{\ell}$. 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 d-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 $\mathsf{validate}(\mathbf{u}_1, \rho \cup \{x\}, \psi)$. This will cause the addition of the target clause $u_1 \vee \overline{x} \vee \bigvee_{\ell \in \rho} \overline{\ell}$. Similarly, either $\mathbf{u}_0$ is a literal node for $\overline{x}$ or $\mathbf{u}_0$ is a product node containing a literal node for $\overline{x}$ as a child. In either case, we recursively call $\mathsf{validate}(\mathbf{u}_0, \rho \cup \{\overline{x}\}, \psi)$, causing the addition of the target clause $u_0 \vee x \vee \bigvee_{\ell \in \rho} \overline{\ell}$. 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 $\mathbf{u}$.
3. If $\mathbf{u}$ is a product node, then we can divide its children into a set of literal nodes L and a set of nonterminal nodes $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k$.
   a. For each literal $\ell \in L$, 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 $\psi|_{\rho \cup \{\overline{\ell}\}}$ 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.
   b. For a single nonterminal child ($k = 1$), we recursively call $\mathsf{validate}\left(\mathbf{u}_1, \rho \cup \bigcup_{\ell \in L} \ell, \psi\right)$.
   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 $\gamma_1, \gamma_2, \ldots, \gamma_k$ 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 $\psi_i = \mathrm{Prov}_\rho(\gamma_i, \psi)$, i.e., those input clauses in ψ that, when simplified, became clause partition γ_i. We recursively call validate $\left(\mathbf{u}_i, \rho \cup \bigcup_{\ell \in L} \ell, \psi_i\right)$.

We then generate the target clause for node $\mathbf{u}$, 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)).

Most 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. 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 $\{\overline{\ell} \mid \ell \in C\}$, 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 force u to evaluate to 0. When marking $\mathbf{u}$, the program adds $\overline{u}$ 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. Appendix B 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 $\ell \in L$ 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 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 (Appendix D) shows 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 [7]. Verification was greatly aided by the Aesop [16] automated tactic. In this section, we briefly describe the functionality we implemented and what we proved about it. More information is provided in Appendix C.

Proof checking. The goal of a CPOG proof is to construct a POG that is equivalent to the input CNF ϕ_I. The checker begins by parsing the input formula, initializing the set of 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 which 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.

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 Var` of propositional formulas. 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 σ `: PropAssignment Var` satisfies φ `: 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 POG definitions formula $\bigwedge_{\mathbf{u} \in P} \theta_u$, 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 φ₁ extends to φ₂ while preserving values on X. -/
def extendsOver (X : Set Var) (φ₁ φ₂ : PropForm Var) :=
  ∀ (σ₁ : PropAssignment Var), σ₁ ⊨ φ₁ → ∃ σ₂, σ₁.agreeOn X σ₂ ∧ σ₂ ⊨ φ₂
/-- Assignments satisfying φ are determined on Y by their values on X. -/
def uniqueExt (X Y : Set Var) (φ : PropForm Var) :=
  ∀ (σ₁ σ₂ : PropAssignment Var), σ₁ ⊨ φ → σ₂ ⊨ φ → σ₁.agreeOn X σ₂ →
    σ₁.agreeOn Y σ₂

invariants.extends_pogDefsForm : extendsOver st.inputCnf.vars ⊤ st.pogDefsForm
invariants.uep_pogDefsForm : uniqueExt st.inputCnf.vars st.allVars st.pogDefsForm
```

The next one guarantees that the set of solutions over the original variables is preserved:

```
def equivalentOver (X : Set Var) (φ₁ φ₂ : PropForm Var) :=
  extendsOver X φ₁ φ₂ ∧ extendsOver X φ₂ φ₁

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 $\theta_u \doteq u \wedge \bigwedge_{\mathbf{v} \in P} \theta_v$:

```
def partitioned : PropForm Var → Prop
  | tr | fls | var _ => True
  | neg φ    => φ.partitioned
  | disj φ ψ => φ.partitioned ∧ ψ.partitioned ∧ ∀ τ, ¬(τ ⊨ φ ∧ τ ⊨ ψ)
  | conj φ ψ => φ.partitioned ∧ ψ.partitioned ∧ φ.vars ∩ ψ.vars = ∅

invariants.partitioned : ∀ u : ILit, (st.pog.toPropForm u).partitioned
invariants.equivalent_lits : ∀ u : ILit, equivalentOver st.inputCnf.vars
    (u ∧ st.pogDefsForm) (st.pog.toPropForm x)
```

These are maintained by valid CPOG proofs. Together with a few additional invariants that ensure the correctness of cached computations, they imply the soundness theorem for P with root node $\mathbf{r}$: the FINAL CONDITIONS ensure that θ is logically equivalent to $\theta_P \doteq \{\{r\}\} \cup \bigcup_{\mathbf{u} \in P} \theta_u$ (`equivalent st.clauseDb (r ∧ st.pogDefsForm)`), and from the invariants we can prove $\phi_I \iff \phi_{\mathbf{r}}$ (`equivalent st.inputCnf (st.pog.toPropForm r)`). After certifying a proof, the checker can pass its in-memory POG representation to the ring evaluator.

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 {φ : PropForm Var} {s : Finset Var} :
  vars φ ⊆ s → partitioned φ → countModels (card s) φ = card (models φ 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 versions of the three programs that, along with the D4 knowledge compiler, form the toolchain illustrated in Figure 1[1]. The proof generator is the same in both cases, since it need not be trusted. Our *verified* versions 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* versions are written in C (checker) and Python (ring evaluator). These are currently faster and more scalable, but we anticipate their 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 [11]. The latter yields shorter proofs

[1] All tools, along with a set of benchmark problems are available at `https://github.com/rebryant/cpog`

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.

Our prototype ring evaluator can perform both standard and weighted model counting. It performs arithmetic over a subset of the rationals we call $\mathbf{Q}_{2,5}$, 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. Appendix D provides a more complete description. For our evaluation, we used the public benchmark problems from the 2022 standard and weighted model competitions. We found that there were 180 unique CNF files among these, ranging in size from 250 to 2,753,207 clauses. 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. We ran our proof generator with a time limit of 10,000 seconds. It 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 generator and checker for a full proof was 13,145 seconds. It is worth noting that, to date, we have not found any errors in the d-DNNF graphs generated by D4.

We found that the monolithic approach for generating the forward implication proof works well for smaller POGs, but it becomes unreliable and inefficient for larger ones. We also found that our two optimizations: literal grouping and lemmas can provide substantial improvements in proof size and runtime. Running the verified proof checker in Lean 4 showed the same scaling with respect to the proof size as did the prototype checker, but it required, on average, around 5.9 times longer.

Finally, 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.

13 Conclusions

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.

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. 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.
4. Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2002.
5. Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *European Conference on Artificial Intelligence*, pages 328–332, 2004.
6. Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 2002.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, July 2017.
14. Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *International Joint Conference on Artificial Intelligence*, pages 667–673, 2017.
15. Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020.
16. 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.
17. John A. Robinson. A machine-oriented logic based on the resolution principle. *J.ACM*, 12(1):23–41, January 1965.
18. 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.
19. 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.
20. Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: reference counting optimized for purely functional programming. In *Implementation and Application of Functional Languages (IFL)*, pages 3:1–3:12. ACM, 2019.

21 Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.

22 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proc. of the 10th Int. Symposium on Artificial Intelligence and Mathematics (ISAIM 2008)*, 2008.

23 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.

24 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.

A CPOG Example

Figure 2 illustrates an example formula and shows how the CPOG file declares its POG representation. The input formula (A) consists of five clauses over variables x_1, x_2, x_3, and x_4. The generated POG (B) has six nonterminal nodes representing four products and two sums. We name these by the node type (product $\mathbf{p}$ or sum $\mathbf{s}$), subscripted by the ID of the extension variable. The first part of the CPOG file (C) declares these nodes using clause IDs that increment by three or four, depending on whether the node has two children or three. The last two nonzero values in the sum declarations are the hint providing the required mutual exclusion proof.

A.1 Node Declarations

We step through portions of the file to provide a better understanding of the CPOG proof framework. Figure 2(D) shows the defining clauses that are implicitly defined by the POG operation declarations. These do not appear in the CPOG file. Referring back to the declarations of the sum nodes in Figure 2(C), we can see that the declaration of node $\mathbf{s}_7$ had clause IDs 7 and 10 as the hint. We can see in Figure 2(A) that these two clauses form a RUP proof for the clause $\overline{p}_5 \vee \overline{p}_6$, showing that the two children of $\mathbf{s}_7$ have disjoint models. Similarly, node $\mathbf{s}_{10}$ is declared as having clause IDs 16 and 19 as the hint. These form a RUP proof for the clause $\overline{p}_8 \vee \overline{p}_9$, showing that the two children of $\mathbf{s}_{10}$ have disjoint models.

A.2 Forward Implication Proof

Figure 2(E) provides the sequence of assertions leading to unit clause 36, consisting of the literal s_{10}. This clause indicates that $\mathbf{s}_{10}$ is implied by the input clauses, i.e., any total assignment α satisfying the input clauses must have $\alpha(s_{10}) = 1$. Working backward, we can see that clause 29 indicates that variable p_8 will be implied by the input clauses when $\alpha(x_1) = 0$, while clause 34 indicates that node p_9 will be implied by the input clauses when $\alpha(x_1) = 1$. These serve as the hint for clause 36.

A.3 Reverse Implication Proof

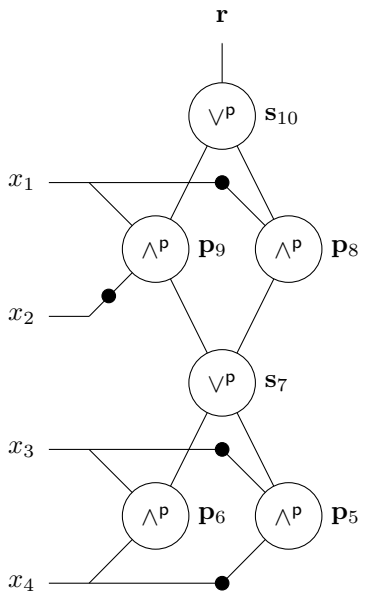
Figure 2(F) shows the RUP proof steps required to delete the input clauses. Consider the first of these, deleting input clause $\overline{x}_1 \vee x_3 \vee \overline{x}_4$. The requirement is to show that there is no total assignment α that falsifies this clause but assigns $\alpha(s_{10}) = 1$. The proof proceeds by first assuming that the clause is false, requiring $\alpha(x_1) = 1$, $\alpha(x_3) = 0$, and $\alpha(x_4) = 1$. The hint then consist of unit clauses (e.g., clause 36 asserting that $\alpha(p_{10}) = 1$) or clauses that cause unit propagation. Hint clauses 8 and 10 force the assignments $\alpha(p_5) = \alpha(p_6) = 0$. These, plus hint clause 12 force $\alpha(s_7) = 0$. This, plus hint clauses 16 and 21 force $\alpha(p_8) = \alpha(p_9) = 0$, leading via clause 22 to $\alpha(s_{10}) = 0$. But this contradicts clause 36, completing the RUP proof. The deletion hints for the other input clauses follow similar patterns—they work from the bottom nodes of the POG upward, showing that any total assignment that falsifies the clause must assign $\alpha(s_{10}) = 0$.

Deleting the asserted clauses is so simple that we do not show it. It involves simply deleting the clauses from clause number 35 down to clause number 25, with each deletion using the same hint as were used to add that clause. In the end, therefore, only the defining clauses for the POG nodes and the unit clause asserting s_{10} remain, completing a proof that the POG is logically equivalent to the input formula.

(A) Input Formula

ID	Clauses	
1	-1 3 -4	0
2	-1 -3 4	0
3	3 -4	0
4	1 -3 4	0
5	-1 -2	0

(B) POG Representation



(C) POG Declaration

ID	CPOG line		Explanation
6	p 5 -3 -4	0	$p_5 = \overline{x}_3 \wedge^p \overline{x}_4$
9	p 6 3 4	0	$p_6 = x_3 \wedge^p x_4$
12	s 7 5 6 7 10	0	$s_7 = p_5 \vee^p p_6$
15	p 8 -1 7	0	$p_8 = \overline{x}_1 \wedge^p s_7$
18	p 9 1 -2 7	0	$p_9 = x_1 \wedge^p \overline{x}_2 \wedge^p s_7$
22	s 10 8 9 16 19	0	$s_{10} = p_8 \vee^p p_9$
	r 10		Root $r = s_{10}$

(D) Implicit Clauses

ID	Clauses		Explanation
6	5 3 4	0	Define p_5
7	-5 -3	0	
8	-5 -4	0	
9	6 -3 -4	0	Define p_6
10	-6 3	0	
11	-6 4	0	
12	-7 5 6	0	Define s_7
13	7 -5	0	
14	7 -6	0	
15	8 1 -7	0	Define p_8
16	-8 -1	0	
17	-8 7	0	
18	9 -1 2 -7	0	Define p_9
19	-9 1	0	
20	-9 -2	0	
21	-9 7	0	
22	-10 8 9	0	Define s_{10}
23	10 -8	0	
24	10 -9	0	

(E) CPOG Assertions

ID	Clause		Hint		Explanation
25	a 5 1 3	0	3 6	0	$\overline{x}_1 \wedge \overline{x}_3 \Rightarrow p_5$
26	a 6 1 -3	0	4 9	0	$\overline{x}_1 \wedge x_3 \Rightarrow p_6$
27	a 3 7 1	0	13 25	0	$\overline{x}_3 \wedge \overline{x}_1 \Rightarrow s_7$
28	a 7 1	0	27 14 26	0	$\overline{x}_1 \Rightarrow s_7$
29	a 8 1	0	28 15	0	$\overline{x}_1 \Rightarrow p_8$
30	a 5 -1 3	0	1 6	0	$x_1 \wedge \overline{x}_3 \Rightarrow p_5$
31	a 6 -1 -3	0	2 9	0	$x_1 \wedge x_3 \Rightarrow p_6$
32	a 3 7 -1	0	13 30	0	$\overline{x}_3 \wedge x_1 \Rightarrow s_7$
33	a 7 -1	0	32 14 31	0	$x_1 \Rightarrow s_7$
34	a 9 -1	0	5 33 18	0	$x_1 \Rightarrow p_9$
35	a 1 10	0	23 29	0	$\overline{x}_1 \Rightarrow s_{10}$
36	a 10	0	35 24 34	0	s_{10}

(F) Input Clause Deletions

CPOG line		Explanation
d 1	36 8 10 12 16 21 22 0	Delete clause 1
d 2	36 7 11 12 16 21 22 0	Delete clause 2
d 3	36 8 10 12 17 19 22 0	Delete clause 3
d 4	36 7 11 12 17 19 22 0	Delete clause 4
d 5	36 16 20 22 0	Delete clause 5

Figure 2 Example formula (A), its POG representation (B), and its CPOG proof (C), (E), and (F)

Observe that the forward implication proof shown in Figure 2 must "visit" nodes $\mathbf{p}_5$ and $\mathbf{p}_6$ twice, separately considering assignments where $\alpha(x_1) = 1$ (clauses 25 and 26) and where $\alpha(x_1) = 1$ (clauses 30 and 31). Section B.3 shows how to use a lemma such that these nodes are each only visited once.

B Optimizations for Forward Implication Proofs

The two optimizations we have implemented exploit the power of our general resolution framework to define new structures within the proof. They create new product nodes that are not part of the POG representation.

B.1 Literal Grouping

A single recursive step of validate can encounter product nodes having many literals as children. The naive formulation of validate considers each literal $\ell \in L$ separately. Literal grouping allows all literals to be validated with a single call to a SAT solver. It collects those literals $\ell_1, \ell_2, \ldots, \ell_m$ that cannot be validated by BCP and defines a product node $\mathbf{v}$ having these literals as children. The goal then becomes to prove that any total assignment must yield 1 for extension variable v. A single call to the solver can generate this proof by invoking it on the formula $\psi|_\rho \cup \theta_v|_{\{\overline{v}\}}$, which should be unsatisfiable. The proof steps can be mapped back into clause addition steps in the CPOG file, incorporating the input clauses and the defining clauses for $\mathbf{v}$ into the hints.

B.2 Lemmas

As we have noted, the recursive calling of validate starting at root $\mathbf{r}$ effectively expands the POG into a tree, and this can lead to an exponential number of calls. These shared subgraphs arise when the knowledge compiler employs *clause caching* to detect that the simplified set of clauses arising from one partial assignment to the literals matches that of a previous partial assignment [4]. When this d-DNNF node is translated into POG node $\mathbf{u}$, the proof generator can assume (and also check), that there is a simplified set of clauses $\gamma_\mathbf{u}$ for which the subgraph with root $\mathbf{u}$ is its POG representation.

The proof generator can exploit the sharing of subgraphs by constructing and proving a *lemma* for each node $\mathbf{u}$ having indegree($\mathbf{u}$) > 1. This proof shows that any total assignment α that satisfies formula $\gamma_\mathbf{u}$ and the defining clauses for the POG must yield $\alpha(u) = 1$. This lemma is then invoked for every node having $\mathbf{u}$ as a child. As a result, the generator will make recursive calls during a call to validate only once for each node in the POG.

The challenge for implementing this strategy is to find a way to represent the clauses for the simplified formula $\gamma_\mathbf{u}$ in the CPOG file. Some may be unaltered input clauses, and these can be used directly. Others, however will be clauses that do not appear in the input formula. We implement these by adding POG product nodes to the CPOG file to create the appropriate clauses. Consider an *argument* clause $C \in \gamma_\mathbf{u}$ with $C = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$. If we define product node $\mathbf{v}$ with arguments $\overline{\ell}_1, \overline{\ell}_2, \ldots, \overline{\ell}_k$, then by Table 2(A), we will introduce a defining clause $v \vee \ell_1 \vee \ell_2 \vee \cdots \ell_k$. We call this a *synthetic* clause having $\overline{v}$ as the *guard literal*. That is, a partial assignment ρ such that $\rho(v) = 0$ will *activate* the clause, causing it to represent argument clause C. On the other hand, a partial assignment with $\rho(v) = 1$ will cause the clause to become a tautology and therefore have no effect.

Suppose for every clause $C_j \in \gamma_\mathbf{u}$ that does not correspond to an input clause, we generate a synthetic clause C'_j with guard literal $\overline{v}_j$, for $1 \leq j \leq m$. Let $\gamma'_\mathbf{u}$ be the formula where each

clause C_j is replaced by synthetic clause C'_j, while input clauses in $\gamma_{\mathbf{u}}$ are left unchanged. Let $\beta = \{\overline{v}_1, \overline{v}_2, \ldots, \overline{v}_m\}$. Invoking $\mathsf{validate}(\mathbf{u}, \beta, \gamma'_{\mathbf{u}})$ will then prove a lemma, given by the target clause $u \vee v_1 \vee v_2 \vee \cdots \vee v_m$, showing that any total assignment α that activates the synthetic clauses will have $\alpha(u) = 1$.

Later, when node $\mathbf{u}$ is encountered by a call to $\mathsf{validate}(\mathbf{u}, \rho, \psi)$, we invoke the lemma by showing that each synthetic clause C_j matches some simplified clause in $\psi|_\rho$. More precisely, for $1 \leq j \leq m$, we use clause addition to assert the clause $\overline{v}_j \vee \bigvee_{\ell \in \rho} \overline{\ell}$, showing that synthetic clause C_j will be activated. Combining the lemma with these activations provides a derivation of the target clause for the call to validate.

Observe that the lemma structure can be hierarchical, since a shared subgraph may contain nodes that are themselves roots of shared subgraphs. Nonetheless, the principles described allow the definition, proof, and applications of a lemma for each shared node in the graph. For any node $\mathbf{u}$, the first call to $\mathsf{validate}(\mathbf{u}, \rho, \psi)$ may require further recursion, but any subsequent call can simply reuse the lemma proved by the first call.

B.3 Lemma Example

Figure 3 shows an alternate forward implication proof for the example of Figure 2 using a lemma to represent the shared node $\mathbf{s}_7$. We can see that the POG with this node as root encodes the Boolean formula $x_3 \leftrightarrow x_4$, having a CNF representation consisting of the clauses $\{x_3, \overline{x}_4\}$ and $\{\overline{x}_3, x_4\}$. The product node declarations shown in Figure 3(A) create synthetic clauses 25 and 28 to encode these arguments with activating literals $\overline{v}_{11}$ and $\overline{v}_{12}$, respectively. Clauses 31–34 then provide a proof of the lemma, stating that any assignment α that activates these clauses will assign 1 to s_7. Clauses 35 and 36 state that an assignment with $\alpha(x_1) = 0$ will cause the first synthetic clause to activate due to input clause 3, and it will cause the second synthetic clause to activate due to input clause 4. From this, clause 37 can use the lemma to state that assigning 0 to x_1 will cause s_7 to evaluate to 1. Similarly, clauses 39 and 40 serve to activate the synthetic clauses when $\alpha(x_1) = 1$, due to input clauses 1 and 2, and clause 41 then uses the lemma to state that assigning 1 to x_1 will cause s_7 to evaluate to 1.

In this example, adding the lemma increases the proof length, but that is only because it is such a simple formula.

C More Information about the Formally Verified Checker

Our verified toolchain has been implemented in Lean 4, which is based on a formal logical foundation in which expressions have a computational interpretation. As with other proof assistants like Isabelle and Coq, a function defined within the formal system can be compiled to efficient code. At the same time, we can state and prove claims about the function within the system, thereby verifying that the functions compute the intended results.

Our code is contained in the directory `ProofChecker`. The main checker loop is implemented in `Checker/CheckerCore.lean`. That file defines a structure `PreState` that contains all the relevant data structures, which include the input CNF formula, the clause database, the POG under construction, and the root literal. We define a `State` of the checker to be a `PreState` that satisfies all the invariants that we need to establish the final result, and the bulk of our work involved showing that these are indeed preserved by each step of the proof, which modifies the clause database and the POG. The ring evaluator and model counter are contained in the `Count` directory. Here we provide a few additional notes on the implementation and clarify the sense in which the toolchain has been verified.

(A) Additional nodes

ID	CPOG line	Explanation
25	`p 11 -3 4 0`	$v_{11} = \overline{x}_3 \wedge^{\mathsf{p}} x_4$
28	`p 12 3 -4 0`	$v_{12} = x_3 \wedge^{\mathsf{p}} \overline{x}_4$

(B) Implicit Clauses

ID	Clauses	Explanation
25	`11 3 -4 0`	Argument clause $\{x_3, \overline{x}_4\}$, activated by $\overline{v}_{11}$
26	`-11 -3 0`	
27	`-11 4 0`	
28	`12 -3 4 0`	Argument clause $\{\overline{x}_3, x_4\}$, activated by $\overline{v}_{12}$
29	`-12 3 0`	
30	`-12 -4 0`	

(C) CPOG Assertions

ID	Clause	Hint	Explanation
Lemma Proof			
31	`a 5 11 12 3 0`	`25 6 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge \overline{x}_3 \Rightarrow p_5$
32	`a 6 11 12 -3 0`	`28 9 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge x_3 \Rightarrow p_6$
33	`a 3 7 11 12 0`	`13 31 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \wedge \overline{x}_3 \Rightarrow s_7$
34	`a 7 11 12 0`	`33 14 32 0`	$(\overline{v}_{11} \wedge \overline{v}_{12}) \Rightarrow s_7$
Lemma Application #1			
35	`a -11 1 0`	`26 27 3 0`	$\overline{x}_1 \Rightarrow \overline{v}_{11}$
36	`a -12 1 0`	`29 30 4 0`	$\overline{x}_1 \Rightarrow \overline{v}_{12}$
37	`a 7 1 0`	`35 36 34 0`	$\overline{x}_1 \Rightarrow s_7$
38	`a 8 1 0`	`37 15 0`	$\overline{x}_1 \Rightarrow p_8$
Lemma Application #2			
39	`a -11 -1 0`	`26 27 1 0`	$x_1 \Rightarrow \overline{v}_{11}$
40	`a -12 -1 0`	`29 30 2 0`	$x_1 \Rightarrow \overline{v}_{12}$
41	`a 7 -1 0`	`39 40 34 0`	$x_1 \Rightarrow s_7$
42	`a 9 -1 0`	`5 41 18 0`	$x_1 \Rightarrow p_9$
43	`a 1 10 0`	`23 38 0`	$\overline{x}_1 \Rightarrow s_{10}$
44	`a 10 0`	`43 24 42 0`	s_{10}

Figure 3 Example of lemma definition, proof, and application

C.1 Implementation notes

When thinking about the formal verification, it is helpful to distinguish between the data structures that play a role in the code that is executed and the definitions that serve as a mathematical reference, allowing us to state our specifications and prove that they are met. Among the former is our representation of CNF formulas: a literal is represented as a nonzero integer, a clause is an array of literals, and a CNF formula is an array of clauses.

```
def ILit := { i : Int // i ≠ 0 }
abbrev IClause := Array ILit
abbrev ICnf := Array IClause
```

We define the clause database `ClauseDb` as a hashmap that stores each clause together with a flag indicating whether it has been deleted. Each element of a POG is either a variable, a binary disjunction (sum), or an arbitrary conjunction (product):

```
inductive PogElt where
  | var  : Var → PogElt
  | disj : Var → ILit → ILit → PogElt
  | conj : Var → Array ILit → PogElt
```

In the first case, the argument `x` in the expression `var x` is the index of an original variable; in `disj x left right` and `conj x args` it is an extension variable appearing in the CPOG file. Note that representing edges as literals allows us to negate the arguments to `disj` and `conj`. A `Pog` is then an array of `PogElt` that is well-founded in the sense that each element depends only on prior elements in the array.

On the mathematical side, we define propositional formulas and their semantics in the usual way. Functions `ILit.toPropForm`, `IClause.toPropForm`, `ICnf.toPropForm`, and `Pog.toPropForm` relate our data structures to propositional formulas and their semantics. For example, given a literal `u`, `Pog.toPropForm P u` denotes the (unnegated or negated) interpretation of the node corresponding to `u` in the POG `P` as a propositional formula $\phi_{\mathbf{u}}/\neg\phi_{\mathbf{u}}$ over the original variables. Lean provides us with convenient "anonymous projector" notation that allows us to write `P.toPropForm u` instead of `Pog.toPropForm P u` when `P` has type `Pog`, `C.toPropForm` instead of `IClause.toPropForm C` when `C` has type `IClause`, etc.

In order to reason about the behavior of the checker, we found it important to work with propositional formulas modulo logical equivalence, a structure known in logic as the *Lindenbaum–Tarski algebra*. We defined the quotient and lifted the Boolean operations and the entailment relation to this new type. The advantage is that equivalent formulas give rise to equal elements in the quotient. This makes it much easier to prove equivalences involving complicated expressions, since it enables us to substitute elements of the quotient for others even when the corresponding formulas are merely equivalent but not equal. This gave rise to auxiliary challenges, however. For example, it is no longer straightforward to say that an element of the quotient "depends" on a variable, since equivalent formulas can have different variables—consider $x \vee \neg x$ and $\top$. Instead, we made use of a semantic notion of dependence, in which an element ϕ of the quotient depends on a variable if and only if for some truth assignment the value of ϕ changes when the assignment to that variable is flipped.

C.2 Trust

In this subsection, we clarify what has been verified and what has to be trusted. Recall that our first step is to parse CNF and CPOG files in order to read in the initial formula and the proof. We do not verify this step. Instead, the verified checker exposes flags `--print-cnf`

and `--print-cpog` which reprint the consumed formula or proof, respectively. Comparing this to the actual files using `diff` provides an easy way of ensuring that what was parsed matches their contents. This involves trusting only the correctness of the print procedure and `diff`. Similarly, if one wants to establish the correctness of the POG contained in the CPOG file, one can print out the POG that is constructed by the checker and compare.

Lean's code extraction replaces calculations on natural numbers and integers with efficient but unverified arbitrary precision versions. Lean also uses an efficient implementation of arrays; within the formal system, these are defined in terms of lists, but code extraction replaces them with dynamic arrays and uses reference counting to allow destructive updates when it is safe to do so [20]. Finally, Lean's standard library implements hashmaps in terms of arrays. Many of the basic properties of hashmaps have been formally verified, but not all. In particular, we make use of a fold operation whose verification is not yet complete. Thus our proofs depend on the assumption that the fold operation has the expected properties.

In summary, in addition to trusting Lean's foundation and kernel checker, we also have to trust that code extraction respects that foundation, that the implementation of the fold operation for hashmaps satisfies its description, and that, after parsing, the computation uses the correct input formula. All of our specifications have been completely proven and verified relative to these assumptions.

D Detailed Experimental Results

Our experiments are designed to evaluate the following:

- whether our toolchain can handle challenging benchmark problems,
- the effectiveness of some design choices and optimizations,
- the comparative performance of the prototype and verified checkers, and
- how the performance of our toolchain compares to that of the MICE model counter verifier.

D.1 Experimental Setup

As a test machine, we used a 2021 MacBook Pro laptop having a 3.2 Ghz Apple M1 processor and with 64 GB of physical memory. We also used an external 3 TB solid-state disk drive with an advertised read/write speed of one gigabit per second. Despite being a laptop, this configuration is somewhat more powerful than the nodes typically found in server clusters. The fast access to storage is especially important due to the very large files generated and processed by the tools. Our tools generated individual files with over 160 gigabytes of data.

For benchmarks, we downloaded 100 files each from the 2022 standard and weighted model counting competitions:

`https://mccompetition.org/2022/mc_description.html`

We found that 20 of these were duplicates across the two tracks, yielding a total of 180 unique benchmark problems, ranging in size from 250 to 2,753,207 clauses.

Our default configuration for the proof generator used the structural method for the forward implication proof, with the two optimizations of literal grouping and lemmas. Running with a time limit of 4000 seconds[2], D4 was able to generate d-DNNF representations for 124 of these, and it timed out on the rest. Our proof generator was able to convert all of the

[2] All measured times in this document are actual elapsed times, not CPU times.

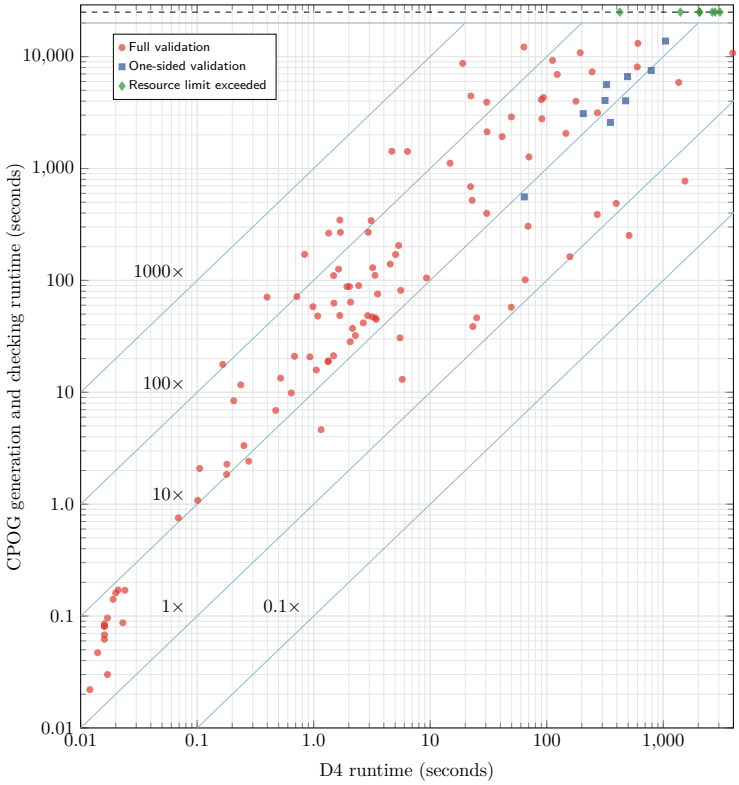


Figure 4 Combined runtime for CPOG proof generation and checking as function of D4 runtime. Timeouts are shown as points on the dashed line.

generated d-DNNF graphs into POGs and determine how many defining clauses they would generate. A POG operation with k arguments requires $k+1$ defining clauses, and so the total number of defining clauses in POG P equals the number of nonterminal nodes plus the number of edges in the graph, corresponding to our definition of $|P|$ in Section 5. The number of defining clauses ranged from 304 to 2,761,457,765 with a median of 848,784.

D.2 Performance of the Proof Generator and Checker

For each of the 124 d-DNNF graphs generated by D4, we ran our proof generator and prototype checker, limiting proof generation to 10,000 seconds. We ran the programs to generate and check one-sided proofs for the graphs, again with a time limit of 10,000 seconds for proof generation.

Figure 4 summarizes our results in terms of the time required by D4 (X axis) versus the sum of the times for the proof generator and checker (Y axis). We were able to complete a full validation for 108 of the 124 benchmarks, with times ranging from fractions of a second to 13,144 seconds, with a median of 71.6 seconds.

Relative to the runtime for D4, two problems ran faster with the generator and checker. These were for problems having small numbers of models (relative to the number of variables), and so most of the time spent by both D4 and our proof generator was in running a CDCL solver to search the very sparse solution space. CaDiCaL generally outperforms the miniSAT-based solver used by D4. At the other extreme, one problem that required only 19 seconds for D4 required 8657 seconds to generate a proof and 45 seconds to check it. This benchmark

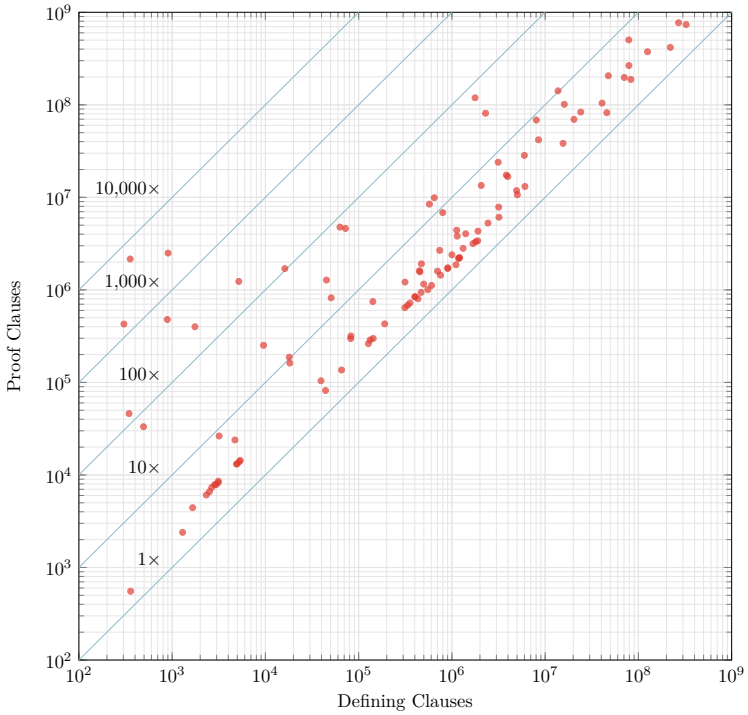


Figure 5 Total number of clauses in CPOG file as function of number of defining clauses

appears to stem from weak performance by our implementation of BCP. Overall the ratios between the combined generation plus checking times versus the time for D4 had a harmonic mean of 5.5.

Of the 16 benchmarks that could not be fully validated, one had so many defining clauses that it overflowed the 32-bit signed integers our programs use for clause identifiers. Another exited due to the virtual memory limit imposed by the operating system, and the other 14 timed out. Of the 15 that did not overflow the clause counter, we were able to complete a one-sided verification for 9, but the other 6 timed out. Overall, we were able to provide some level of verification for 94% of the benchmark problems.

Figure 5 compares the number of defining clauses (X axis) with the total number of clauses (Y axis) for the 108 problems that were fully verified. These totals include the defining clauses for the POG, the additional defining clauses introduced to support literal grouping and lemmas, and the clauses added by RUP steps in the forward implication proof. These totals ranged from 554 to 770,382,773, with a median of 1,719,245. The ratios of the total clauses versus defininging clauses ranged from 1.54 to 6073, with a harmonic mean of 3.13. The high numbers were for problems having very few models, and so the proof clauses must encode large proofs of unsatisfiability.

D.3 Monolithic Forward Implication Proofs

To assess the relative merits of the two approaches to forward implication proof generation, we ran the proof generator in monolithic mode on the 92 problems for which the combination of structural proof generation and checking required at most 3000 seconds and then kept only those measurements for which at least one of the two approaches had combined times (generation plus checking) of at most 1000 seconds. There were a total of 83 such problems.

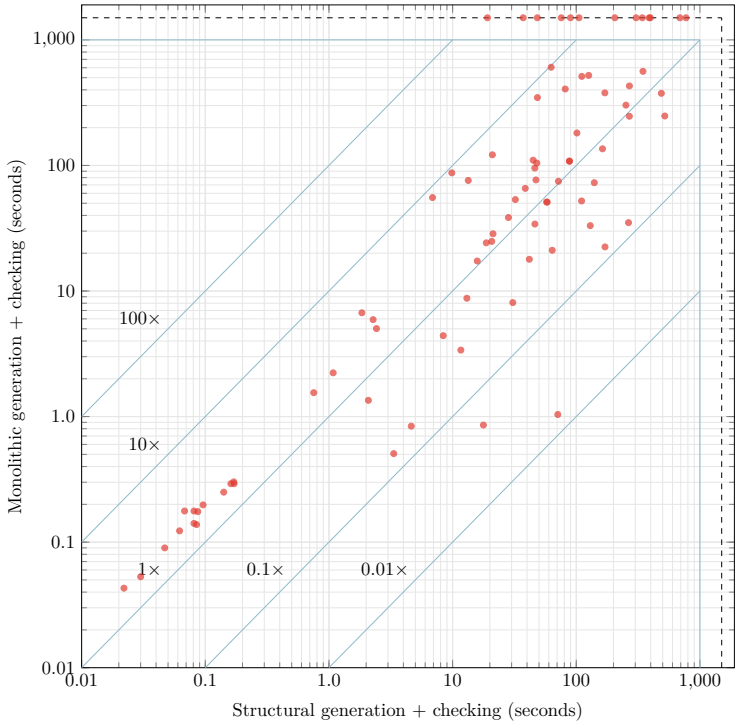


Figure 6 Monolithic versus Structural Validation: Time. Timeouts are shown as points on the dashed lines.

Figures 6 and 7 show comparisons between the two approaches in terms of time and total clauses.

Examining Figure 6, we can see that the monolithic approach ran faster than the structural approach for 23 of the problems, including a case where the generator and checker required only one second, versus 71 seconds for the structural approach. On the other hand, the monolithic approach timed out for 13 of the cases and was slower for the other 37. In general, we can see the monolithic approach faring worse on larger problems.

Figure 7 shows the total number of clauses for the two approaches for the 70 cases where both approaches completed within 1000 seconds. As can be seen, the monolithic approach consistently produces shorter proofs, perhaps due to the benefits of the proof trimming by DRAT-TRIM.

These experiments suggest that a hybrid of the two approaches might yield the best results in terms of consistency, runtime, and proof size. It would begin the recursive descent of validate according to a structural approach, but monolithically generate the proof once it encounters a sufficiently small subgraph.

D.4 Impact of Optimizations

To assess the impact of the two optimizations: literal grouping and lemmas, we ran the proof generator on the 80 problems for which the combination of the two optimizations yielded proofs totaling at most 10 million clauses. Compared to the problems used to assess monolithic proof generation (Section D.3), this set included one that had a short proof but long runtime and excluded four that had short runtimes but long proofs. We then ran the proof generator on all three combinations of the optimizations being partially or totally

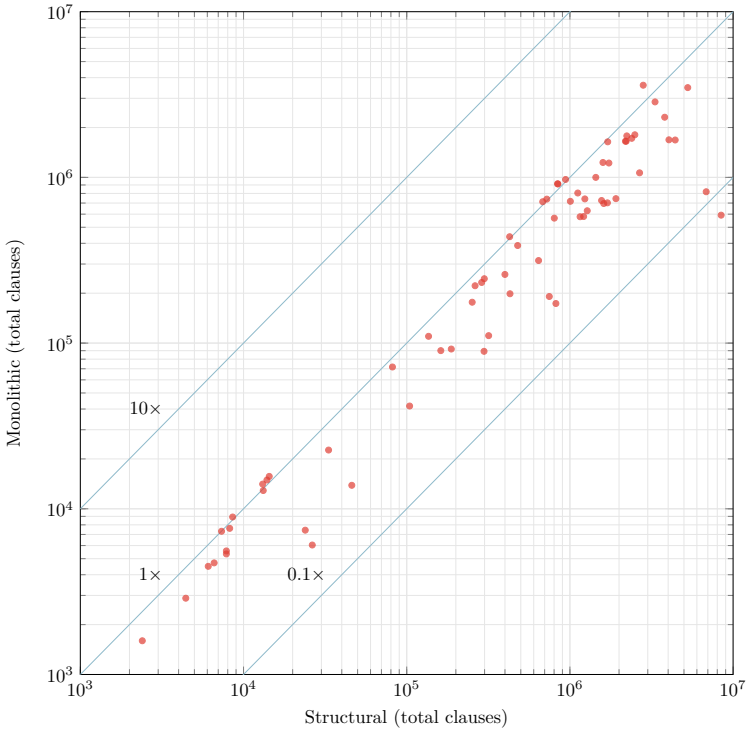


Figure 7 Monolithic versus Structural Validation: Clauses

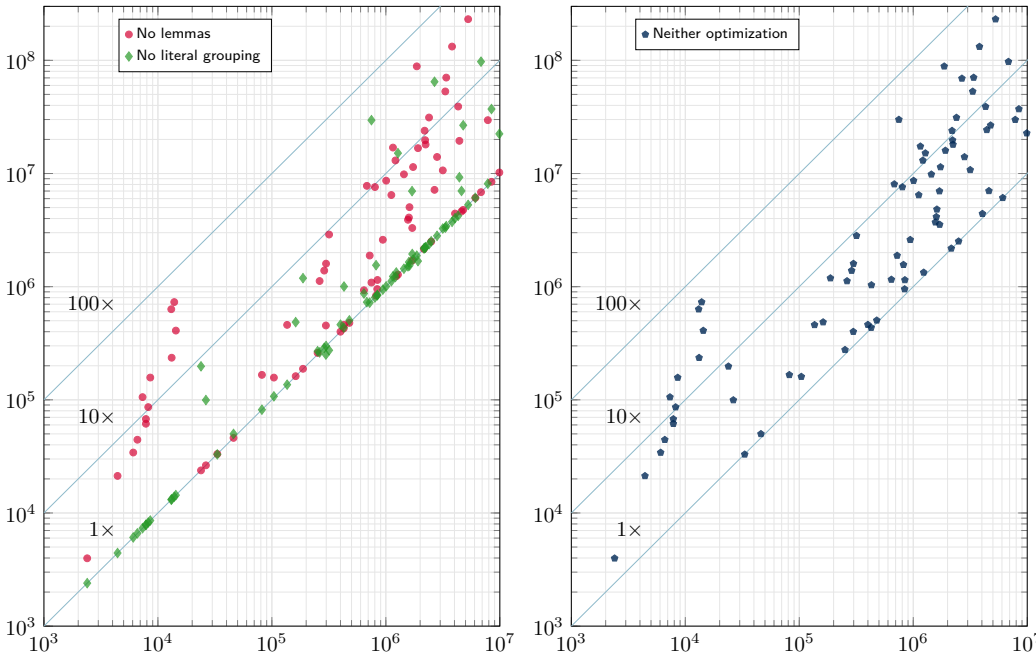


Figure 8 Proof clauses when one (left) or both (right) optimizations are disabled, versus when both optimizations are enabled

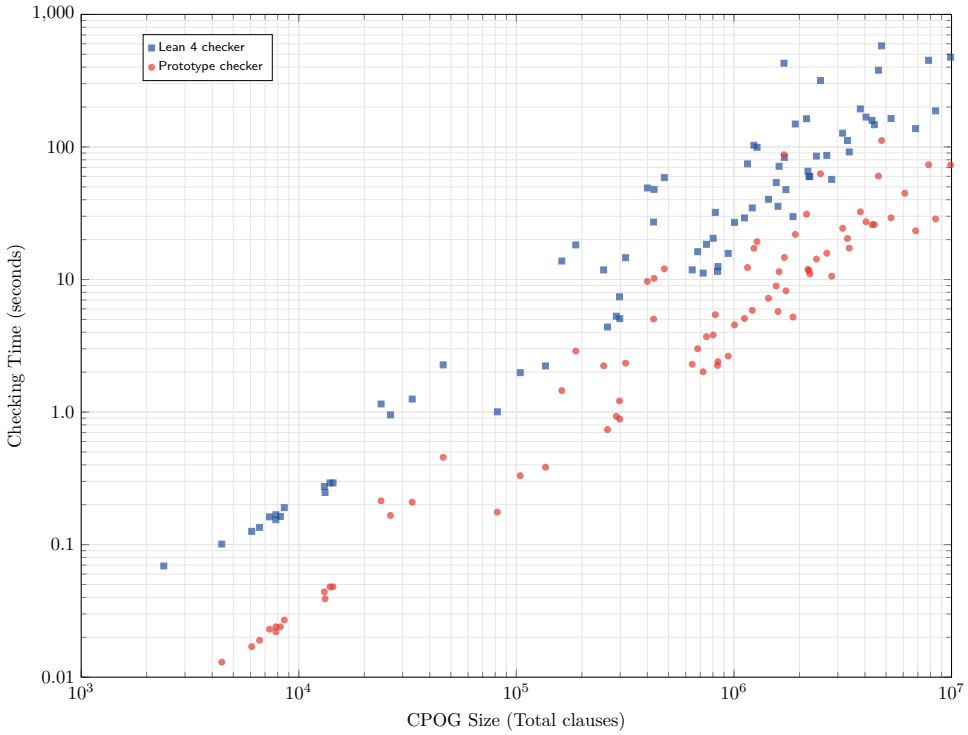


Figure 9 Times for Prototype and Verified Checker, as Functions of Total Clauses

disabled. Figure 8 shows how disabling these optimizations affected the total number of clauses in the proofs, where the X axis indicates the total number with both optimizations enabled, while the Y axis indicates the number with one (left) or both (right) optimizations disabled.

As the figures illustrated, each optimization on its own can shorten the proofs, sometimes substantially, and the two combine to have even greater effect. Of the 80 benchmark problems evaluated, and comparing with both optimizations disabled, 47 had the number of proof clauses reduced by a factor of 2 by using lemmas, 14 by using literal merging, and 60 by enabling both optimizations. 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 causes another proof to grow by a factor of 39.6.

The time performance of these optimizations is less dramatic, but still significant. Lemmas sometimes made proof generation slower, but in one case ran 38× faster. Similarly, literal grouping could take longer, but in one case ran 3.9× faster.

Overall, it is clear that these two optimizations are worthwhile, and sometimes critical, to success for some benchmarks.

D.5 Evaluation of the Verified Checker

We ran the Lean 4 checker on the 80 benchmark problems for which the generated CPOG file had at most 10 million total clauses. These are the same benchmarks used in evaluating the optimizations (Section D.4). All of the checks completed and confirmed the outcome of the prototype checker.

Figure 9 shows the performance of both the Lean 4 checker and the prototype checker on

the Y axis, as functions of the total number of clauses on the X axis. As would be expected, the Lean 4 checker consistently runs slower than the prototype checker. The ratio of the runtime for Lean 4 versus the runtime for the prototype checker had a harmonic mean of 5.94.

Importantly, however, it can be seen that both checkers show the same overall performance trends. The ratio of the two runtimes was less than 8.0 for all but three small benchmarks. This seems like a reasonable price to pay for a rigorous guarantee of correctness, and we are confident that we can reduce this gap with more effort in optimizing the verified checker.

D.6 Comparison with the MICE Framework

To compare the performance of our toolchain versus that of MICE [8], we consider how the combination of proof generation, proof checking, and ring evaluation in the CPOG toolchain compares to the time to run their proof generator NNF2TRACE and checker SHARPTRACE. We add the time for ring evaluation on our side, since their checker provides a count, although only an unweighted one. We used the prototype versions of the checker and ring evaluator. We ran their tools on the 92 benchmark problems for which our toolchain requires at most 3000 seconds and retained the 84 data points for which at least one of the toolchains completes in under 1000 seconds.

We can summarize the results as:

- 84 total data points
- 76 were completed by both programs in under 1000 seconds
- Of those 21 were faster with MICE, 55 with CPOG
- 7 completed with CPOG under 1000 seconds but exceeded that threshold for MICE
- 1 completed with MICE under 1000 seconds but exceeded that threshold for CPOG

The detailed results are shown in Figure 10. As can be seen, the comparative runtimes are highly variable, reflecting the fact that the two toolchains differ fundamentally in their objectives and their approaches.

One shortcoming of NNF2TRACE can be seen in the near-vertical series of points in the upper-lefthand corner. These correlate with a similar set of points along the left in Figure 8. Like the naive implementation of validate, their program recursively traverses the graph representation of a formula, effectively expanding it into a tree. They lack an optimization analogous to lemmas.

We have not tried the MICE toolchain on larger benchmarks, but we suspect it would encounter significant scaling limitations.

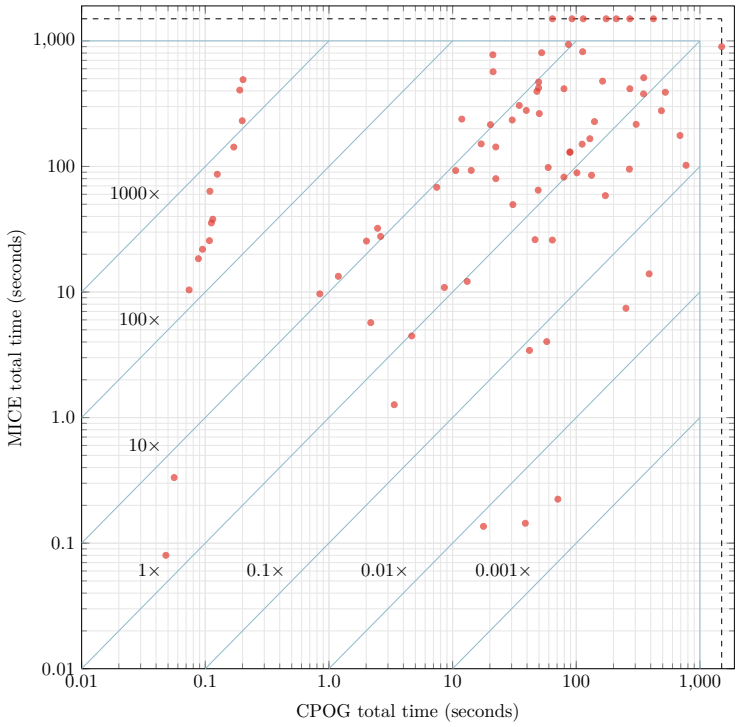


Figure 10 Running Time for MICE versus CPOG proof chains. Timeouts are shown as points on the dashed lines.