Certified Knowledge Compilation
with Application to Formally Verified Model Counting

Randal E. Bryant   REBRYANT@CMU.EDU
Wojciech Nawrocki   WJNAWROCKI@CMU.EDU
Jeremy Avigad   AVIGAD@CMU.EDU
Marijn J. H. Heule   MARIJN@CMU.EDU
Carnegie Mellon University
Pittsburgh, PA USA

Abstract

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

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

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

1. Introduction

Given a Boolean formula, modern Boolean satisfiability (SAT) solvers can find an assignment satisfying it 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 extensions of 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 (Valiant, 1979). 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 (2001), as well as the more restricted *decision decomposable negation normal form* (decision-DNNF) (Huang & Darwiche, 2007; Beame, Li, Roy, & Suciu, 2013) represent a Boolean formula as a directed acyclic graph, with terminal nodes labeled by Boolean variables and their complements, and with each nonterminal node labeled by a Boolean AND or OR operation. Restrictions are placed on the structure of the graph (described in Section 5) such that a count of the models can be computed by a single bottom-up traversal. Kimmig, den Broeck, and Raedt (2017) present a very general *algebraic model counting* framework describing properties of Boolean functions that can be efficiently computed from a d-DNNF representation. These include unweighted and weighted model counting, and much more.

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

This paper introduces *Partitioned-Operation Graphs* (POGs), a form that can encode all of the representations produced by current knowledge compilers. The CPOG (for "certified" POG) file format then captures both the structure of a POG and a checkable proof of its logical equivalence to a Boolean formula in conjunctive normal form (CNF). A CPOG proof consists of a sequence of clause addition and deletion steps, based on an extended resolution proof system (Tseitin, 1983). We establish a set of conditions that, when satisfied by a CPOG file, guarantees that it encodes a well-formed POG and provides a valid equivalence proof.

Figure 1 illustrates our certifying knowledge compilation and model counting toolchain. Starting with input formula ϕ_I, the D4 knowledge compiler (Lagniez & Marquis, 2017) generates a decision-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 an unweighted 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 (de Moura & Ulrich, 2021). Running these two programs on a CPOG file gives strong assurance that the proof and the
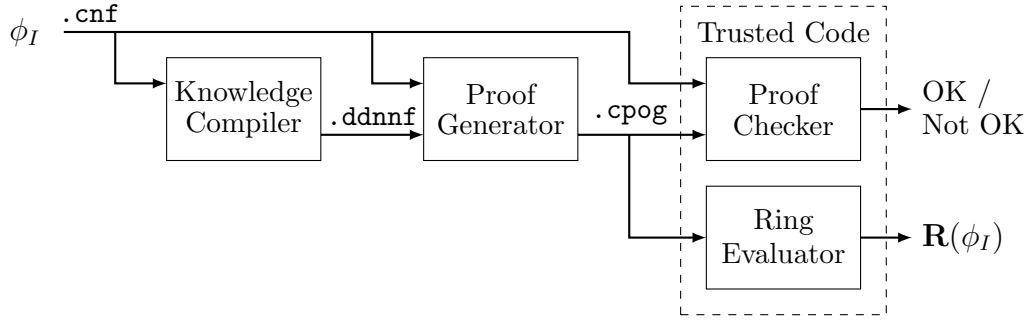
Figure 1: Certifying toolchain. The ring evaluator produces a weighted or unweighted count. Certification by the proof checker guarantees its correctness.

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, as described in Section 10, our efforts to formally verify our proof framework exposed subtle conditions that we had to impose on our partitioned sum rule.

We evaluate our toolchain using benchmark formulas from the 2022 unweighted and weighted model competitions. Our tools handle all but the largest graphs generated by D4. We evaluate the benefits of several optimizations, finding that the use of lemmas to exploit the sharing of subgraphs in the decision-DNNF representation can be critical to avoid an expansion of the graph into a tree. We measure the relative performance of the verified checker with one designed for high performance and capacity, finding that the time to run the verified checker remains within a factor of $4\times$ that of the high capacity checker for most benchmarks, and that it has similar scaling properties. We also show that our tools can provide end-to-end verification of formulas that have been transformed by an equivalence-preserving preprocessor. That is, verification is based on the original formula, and so proof checking certifies correct operation of the preprocessor, the knowledge compiler, and the proof generator.

Our current tool can only handle the representations generated by the D4 knowledge compiler, and it only supports a subclass of the Boolean function properties enabled by algebraic model counting (Kimmig et al., 2017). Both of these shortcomings can be overcome by modest extensions, as discussed in Section 13.

This paper is an extended version of one published at the 2023 Conference on the Theory and Application of Boolean Satisfiability (Bryant, Nawrocki, Avigad, & Heule, 2023). It provides much greater detail about the algorithms, the formal verification, and the experimental results.

2. Related Work

Generating proofs of unsatisfiability in SAT solvers has a long tradition (Zhang & Malik, 2003) and has become widely accepted due to the formulation of clausal proof systems for which proofs can readily be generated and efficiently checked (Heule, Hunt, Jr., & Wetzler,

2013b; Wetzler, Heule, & Hunt Jr., 2014). A number of formally verified checkers have been developed within different verification frameworks (Cruz-Filipe, Heule, Hunt, Kaufmann, & Schneider-Kamp, 2017; Heule, Hunt, Kaufmann, & Wetzler, 2017; Lammich, 2020; Tan, Heule, & Myreen, 2021). The associated proofs add clauses while preserving satisfiability until the empty clause is derived. Our work builds on the well-established technology and tools associated with clausal proof systems, but we require features not found in proofs of unsatisfiability. In particular, our checker constructs an entirely new representation of the input formula. The proof must demonstrate that the new representation satisfies a set of rules, and that it is logically equivalent to the input formula. This requires verifying additional proof steps, including clause deletion steps, and subtle invariants, as described in Sections 7 and 10.

Capelli (2019) and Capelli, Lagniez, and Marquis (2021) developed a knowledge compiler that generates a certificate in a proof system that is itself based on decision-DNNF. Their CD4 program, a modified version of D4, generates annotations to the compiled representation, providing information about how the compiled version relates to the input clauses. It also generates a file of clausal proof steps in the DRAT format (Wetzler et al., 2014). Completing the certification involves running two different checkers on the annotated decision-DNNF graph and the DRAT file. Although the authors make informal arguments regarding the soundness of their frameworks, these do not provide strong levels of assurance. Indeed, we have identified a weakness in their methodology due to an invalid assumption about the guarantees provided by DRAT-TRIM, the program it uses to check the DRAT file. This weakness is *exploitable*: their framework can be "spoofed" into accepting an incorrect compilation.

In more detail, CD4 emits a sequence of clauses R that includes the conflict clauses that arose during a top-down processing of the input clauses. Given input formula ϕ_I, their first task is to check whether $\phi_I \Rightarrow R$, i.e., that any assignment that satisfies ϕ_I also satisfies each of the clauses in R. They then base other parts of their proof on that property and use a separate program to perform a series of additional checks. They use DRAT-TRIM to prove the implication, checking that each clause in R satisifies the *resolution asymmetric tautology* (RAT) property with respect to the preceding clauses (Järvisalo, Heule, & Biere, 2012; Heule et al., 2013b). Adding a RAT clause C to a set of clauses maintains satisfiability, i.e., it will not cause a satisfiable formula to become unsatisfiable. On the other hand, it does not necessarily preserve models, i.e., it can exclude some previous satisfying assignments. As an example, consider the following formulas over the variables x_1, x_2, and x_3:

$$\begin{array}{ll} \phi_1\colon & (x_1 \vee x_3) \\ \phi_2\colon & (x_1 \vee x_3) \quad \wedge \quad (x_2 \vee \overline{x}_3) \end{array}$$

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

This weakness would allow a buggy (or malicious) version of CD4 to spoof the checking framework. Given formula ϕ_1 as input, it could produce a compiled result, including annotations, based on ϕ_2 and also include the second clause of ϕ_2 in R. The check with

DRAT-TRIM would pass, as would the other tests performed by their checker. We have confirmed this possibility with their compiler and checker.[1]

This weakness can be corrected by restricting DRAT-TRIM to only allow adding clauses that obey the stronger *reverse unit propagation* (RUP) property (Goldberg & Novikov, 2003; Van Gelder, 2008). Adding a RUP clause C to a set of clauses does not change the set of satisfying assignments. We have added a command-line argument to DRAT-TRIM that enforces this restriction.[2] This weakness, however, illustrates the general challenge of developing a new proof framework. As we can attest, without engaging in an effort to formally verify the framework, there are likely to be conditions that make the framework unsound.

Fichte, Hecher, and Roland (2022) devised the MICE proof framework for model counting programs. Their proof rules are based on the algorithms commonly used by model counters. They developed a program that can generate proof traces from decision-DNNF graphs and a program to check adherence to their proof rules. This framework is not directly comparable to ours, since it only certifies the unweighted model count, but it has similar goals. Again, they provide only informal arguments regarding the soundness of their framework.

Both of these prior certification frameworks are strongly tied to the algorithms used by the knowledge compilers and model counters. Some of the conditions to be checked are relevant only to specific implementations. Our framework is very general and is based on a small set of proof rules. It builds on the highly developed concepts of clausal proof systems. These factors were important in enabling formal verification. In Section 12, we also compare the performance of our toolchain to these other two. We find that the CD4 toolchain generally outperforms ours, handling many cases where our toolchain cannot completed within the time limit. On the other hand, we find that the MICE toolchain does not scale as well as ours, especially for decision-DNNF graphs with extensive sharing among the subgraphs.

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. *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. Downloaded May 18, 2023 as
   https://github.com/crillab/d4/tree/333370cc1e843dd0749c1efe88516e72b5239174.
2. Available at https://github.com/marijnheule/drat-trim/releases/tag/v05.22.2023.

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 $\phi = \bigwedge_{1 \le i \le k} \phi_i$ *is a Boolean formula, with* $\mathcal{D}(\phi) = \bigcup_{1 \le i \le k} \mathcal{D}(\phi_i)$ *and* $\mathcal{M}(\phi) = \bigcap_{1 \le i \le k} \mathcal{M}(\phi_i)$.

5. *For formulas* $\phi_1, \phi_2, \ldots, \phi_k$, *their* sum $\phi = \bigvee_{1 \le i \le k} \phi_i$ *is a Boolean formula, with* $\mathcal{D}(\phi) = \bigcup_{1 \le i \le k} \mathcal{D}(\phi_i)$ *and* $\mathcal{M}(\phi) = \bigcup_{1 \le i \le k} \mathcal{M}(\phi_i)$.

We highlight some special classes of Boolean formulas. A formula is in *negation normal form* (NNF) when negation is applied only to variables. A formula is in *conjunctive normal form* (CNF) when (i) it is in negation normal form, (ii) sum is applied only to literals, and (iii) there is a single product operation over all of the sums. 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. *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* $\bigwedge_{1 \le i \le k} \phi_i$ *requires* $\mathcal{D}(\phi_i) \cap \mathcal{D}(\phi_j) = \emptyset$ *for* $1 \le i < j \le k$.

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

We let $\wedge^{\mathsf{p}}$ and $\vee^{\mathsf{p}}$ denote the product and sum operations in a partitioned-operation formula. In the knowledge compilation literature, Boolean formulas where all product arguments have disjoint dependency sets are said to be *decomposable* (Darwiche, 2001; Darwiche & Marquis, 2002). Those where all sum arguments have disjoint models are said to be *deterministic* (Darwiche, 2002; Darwiche & Marquis, 2002).

4. Ring Evaluation of a Boolean Formula

We propose a general framework for summarizing properties of Boolean formulas similar to the formulation of algebraic model counting by Kimmig et al. (2017). Our formulation in terms of rings is more restrictive than their semiring-based approach. We discuss the difference and how our work could be generalized in Section 13.2.

Definition 3. *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*

$$\boldsymbol{R}(\phi, w) \;\; = \;\; \sum\nolimits_{\alpha \in \mathcal{M}(\phi)} \;\; \prod\nolimits_{\ell \in \alpha} w(\ell) \tag{1}$$

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 (unweighted) *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.

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 $x \in X$, choose a value from $\mathcal{A}$ at random for $w(x)$. Two formulas ϕ_1 and ϕ_2 will clearly have $\mathbf{R}(\phi_1, w) = \mathbf{R}(\phi_2, w)$ if they are logically equivalent, and if $\mathbf{R}(\phi_1, w) \neq \mathbf{R}(\phi_2, w)$, then they are clearly inequivalent. If they are not equivalent, then the probability that $\mathbf{R}(\phi_1, w) \neq \mathbf{R}(\phi_2, w)$ will be at least $\left(1 - \frac{1}{m}\right)^n \geq \left(1 - \frac{1}{2n}\right)^n > 1/2$. Function hashing can therefore be used as part of a randomized algorithm for equivalence testing (Blum, Chandra, & Wegman, 1980). 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 1. *Ring evaluation with operations* $\neg$, $\wedge^p$, *and* $\vee^p$ *satisfies the following for any weight function* w:

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

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

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

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

A *partitioned-operation graph* (POG) is a directed, acyclic graph with nodes N and edges $E \subseteq N \times N$. We denote nodes with boldface symbols, such as $\mathbf{u}$ and $\mathbf{v}$. When $(\mathbf{u}, \mathbf{v}) \in E$, node $\mathbf{v}$ is said to be a *child* of node $\mathbf{u}$. The in- and out-degrees of node $\mathbf{u}$ are defined as $\text{indegree}(\mathbf{u}) = |E \cap (N \times \{\mathbf{u}\})|$, and $\text{outdegree}(\mathbf{u}) = |E \cap (\{\mathbf{u}\} \times N)|$. Node $\mathbf{u}$ is said to be *terminal* if $\text{outdegree}(\mathbf{u}) = 0$. A terminal node is labeled by a Boolean constant or variable. Node $\mathbf{u}$ is said to be *nonterminal* if $\text{outdegree}(\mathbf{u}) > 0$. A nonterminal node is labeled by Boolean operation $\wedge^p$ or $\vee^p$. A node can be labeled with operation $\wedge^p$ or $\vee^p$ only if it satisfies the partitioning restriction for that operation. Every POG has a designated *root node* $\mathbf{r}$. Each edge has a *polarity*, indicating whether (negative polarity) or not (positive polarity) the corresponding argument should be negated.

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

We can now define and compare two related representations:

- A d-DNNF graph can be viewed as a POG with negation applied only to variables.

- A decision-DNNF graph is a d-DNNF graph with the further restriction that any sum node $\mathbf{u}$ has exactly two children $\mathbf{u}_1$ and $\mathbf{u}_0$, and it has an associated *decision variable* x. For $b \in \{0, 1\}$, node $\mathbf{u}_b$ can be a terminal node with variable x, where the polarity of the edge from $\mathbf{u}$ to $\mathbf{u}_b$ is negative for $b = 0$ and positive for $b = 1$. Alternatively, $\mathbf{u}_b$ can be a product node having either literal $\overline{x}$ ($b = 0$), or literal x ($b = 1$) as one of its arguments. Either form implies that any total assignment $\alpha \in \mathcal{M}(\phi_{\mathbf{u}_b})$ has $\alpha(x) = b$, for $b \in \{0, 1\}$.

The generalizations encompassed by POGs have also been referred to as *deterministic decomposable circuits* (d-Ds) (Monet & Olteanu, 2018). Our current proof generator only works for knowledge compilers generating decision-DNNF representations, but these generalizations allow for future extensions, while maintaining the ability to efficiently perform ring evaluation. Extending the tool to handle arbitrary POGs is discussed in Section 13.1.

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

6. Clausal Proof Framework

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 state 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 (Tseitin, 1983), where proof steps can introduce new *extension variables* encoding Boolean formulas over input and prior extension variables. That is, each extension variable z is introduced via a set of *defining clauses* that encode a formula $z \Leftrightarrow F$, where F is a Boolean formula over a subset of the input variables X and previously defined extension variables Z. We write θ for formulas encoded as clauses, possibly with extension variables, and ϕ for formulas that use no extension variables.

Let $Z(\theta)$ denote the set of extension variables occurring in formula θ. For any total assignment α to the variables in X, the defining clauses induce a unique assignment α^* to the variables in $X \cup Z(\theta)$. For Boolean formula ϕ over variables X and clausal formula θ over the variables $X \cup Z(\theta)$, we say that ϕ is *equivalent over X* to θ, written $\phi \Leftrightarrow_X \theta$, when for any assignment α to the variables in X, assignment α is a model of ϕ if and only if its extension α^* is a model of θ. Starting with θ equal to input formula ϕ_I, the proof must maintain the invariant that $\phi_I \Leftrightarrow_X \theta$.

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 (Goldberg & Novikov, 2003; Van Gelder, 2008). RUP is the core rule supported by standard proof checkers (Heule et al., 2013b; Wetzler et al., 2014) for propositional logic. It provides a simple and efficient way to check a sequence of applications of the resolution proof rule (Robinson, 1965). 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 $\bigwedge_{1 \leq i \leq k} D_i \Rightarrow C$ by showing that the combination of the antecedents plus the negation of C leads to a contradiction. The negation of C is the formula $\bar{\ell}_1 \wedge \bar{\ell}_2 \wedge \cdots \wedge \bar{\ell}_p$, having a CNF representation consisting of p unit clauses of the form $\bar{\ell}_i$ for $1 \leq i \leq p$. A RUP check processes the clauses of the antecedent in sequence, inferring additional unit clauses. In processing clause D_i, if all but one of the literals in the clause is the negation of one of the accumulated unit clauses, then we can add this literal to the accumulated set. That is, all but this literal have been falsified, and so it must be set to true for the clause to be satisfied. The final step with clause D_k must cause a contradiction, i.e., all of its literals are falsified by the accumulated unit clauses.

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

causes a satisfiable set of clauses to become unsatisfiable, and therefore it need only justify clause additions.

7. The CPOG Representation and Proof System

A CPOG file provides both a declaration of a POG, as well as a checkable proof that a Boolean formula, given in conjunctive normal form, is logically equivalent to the POG. The proof format draws its inspiration from the LRAT (Heule et al., 2017) and QRAT (Heule, Seidl, & Biere, 2014) 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. This is the only means for adding extension variables to the proof.

- 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 being the IDs of added clauses. Clause identifiers must be defined in order, with any clause identifier C' given in the hint when adding clause C having $C' < C$.

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`).

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

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

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

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 $\vee^\mathsf{p}$

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

The declaration of a *product* operation, creating a node with operation $\wedge^\mathsf{p}$, 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, indicating the arguments as literals of existing variables. As Table 2(A) shows, this declaration implicitly causes $k+1$ clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the product of its arguments.

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

The declaration of a *sum* operation, creating a node with operation $\vee^\mathsf{p}$, 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, indicating the arguments as literals of existing variables. Hint H consists of a sequence of clause IDs, all of which must be defining clauses for other POG operations.[3] As Table 2(B) shows, this declaration implicitly causes three clauses to be added to the proof, providing a Tseitin encoding that defines extension variable

---

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

v as the sum of its arguments. The hint must provide a RUP proof of the clause $\bar{\ell}_1 \vee \bar{\ell}_2$, showing that the two children of this node have disjoint models.

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

7.2 Semantics

As was described in Section 6, the defining clauses for the product and sum operations uniquely define the values of their extension variables for any assignment of values to the argument variables. That is, for assignment α to the variables in X, the defining clauses induce a unique assignment α^* to all data and extension variables. Every POG node $\mathbf{u}$ represents POG formula $\phi_{\mathbf{u}}$ and has an associated extension variable u. We can prove that for any total assignment α to the input variables, we will have $\alpha^*(u) = 1$ if and only if $\alpha \in \mathcal{M}(\phi_{\mathbf{u}})$.

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

1. There is exactly one remaining clause that was added via RUP addition, and this is a unit clause consisting of root literal r.

2. All of the input clauses have been deleted.

In other words, at the end of the proof it must hold that the active clauses be exactly those in $\theta_P \doteq \{\{r\}\} \cup \bigcup_{\mathbf{u} \in P} \theta_u$, the formula consisting of unit clause $\{r\}$ and the defining clauses for the nodes, providing a Tseitin encoding of P. By our invariant, we are guaranteed that $\phi_I \Leftrightarrow_X \theta_P$. That is, for any total assignment α to the input variables, α is in $\mathcal{M}(\phi_I)$ if and only if its unique extension α^* to the POG variables satisfies $\alpha^*(r) = 1$.

The sequence of clause addition steps provides a *forward implication* proof that $\alpha \in \mathcal{M}(\phi_I) \Rightarrow \alpha^*(r) = 1$. That is, any total assignment α satisfying the input formula must, when extended, also satisfy the formula represented by the POG. Conversely, the sequence of clause deletion steps that delete all intermediate added clauses and all input clauses provides a *reverse implication* proof: $\alpha^*(r) = 1 \Rightarrow \alpha \in \mathcal{M}(\phi_I)$. It does so by contradiction, proving that when $\alpha^*(r) = 0$, we must have $\alpha \notin \mathcal{M}(\phi_I)$.

7.3 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 each sum declaration is the hint providing the required mutual exclusion proof.
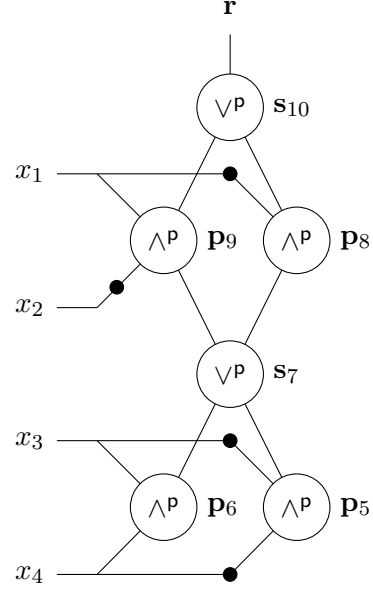
(A) Input Formula

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

(D) Defining 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). The defining clauses (D) are implicitly defined by the POG declaration (C).

7.4 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 $\mathbf{s}_7$ has clause IDs 7 and 10 as the hint. We can see in Figure 2(D) that these two clauses form a RUP proof for the clause $\bar{p}_5 \vee \bar{p}_6$, 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 $\bar{p}_8 \vee \bar{p}_9$, showing that the two children of $\mathbf{s}_{10}$ have disjoint models.

7.5 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 its extension to α^* yield $\alpha^*(s_{10}) = 1$. Working backward, we can see that clause 35 indicates that variable s_{10} will be implied by the input clauses when $\alpha(x_1) = 0$. Clause 34 indicates that node p_9 will be implied by the input clauses when $\alpha(x_1) = 1$, while defining clause 24 shows that node s_{10} will be implied by the input clauses when $\alpha^*(p_9) = 1$. These three clauses serve as the hint for clause 36.

7.6 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 $\bar{x}_1 \vee x_3 \vee \bar{x}_4$. The requirement is to show that there is no total assignment α that falsifies this clause but extends to α^* such that $\alpha^*(s_{10}) = 1$. 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 consists of unit clauses (e.g., clause 36 asserting that $\alpha^*(s_{10}) = 1$) 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, when extended, have $\alpha^*(s_{10}) = 0$.

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 was 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.

8. Generating CPOG from Decision-DNNF

A decision-DNNF graph can be directly translated into a POG. In doing this conversion, our program performs simplifications to eliminate Boolean constants. Except in degenerate cases, where the formula is unsatisfiable or a tautology, we can therefore assume that the POG does not contain any constant nodes. In addition, negation is only applied to variables, and so the only edges with negative polarity will have variables as children. We can therefore

view the POG as consisting of *literal* nodes corresponding to input variables and their negations, along with *nonterminal* nodes, which can be further classified as *product* and *sum* nodes.

8.1 Forward Implication Proof

For input formula ϕ_I and its translation into a POG P with root node $\mathbf{r}$, the most challenging part of the proof is to show that $\mathcal{M}(\phi_I) \subseteq \mathcal{M}(\phi_{\mathbf{r}})$, i.e., that any total assignment α that is a model of ϕ_I will extend to assignment α^* such that $\alpha^*(r) = 1$, 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 for 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 monolithic approach is *logically complete*, i.e., assuming the CNF formula is equivalent to the POG, and given enough time and computing resources, it can generate a CPOG proof of equivalence. The *structural* approach only works when the POG was generated from a decision-DNNF graph having a structure that reflects the top-down process by which it was created. It recursively traverses the POG, generating proof obligations at each node encountered. It may require multiple calls to a proof-generating SAT solver.

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

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 general use.

The structural approach to proof generation takes the form of 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 extend to assignment α^* such that $\alpha^*(u) = 1$. 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 $\mathsf{validate}(\mathbf{u}, \rho, \psi)$ assumes that we have traversed a path from the root node down to node $\mathbf{u}$, with the literals encountered in the product nodes forming the partial assignment ρ. The set of clauses ψ can be a proper subset of the input clauses ϕ_I when a product node has caused a splitting into clauses containing disjoint variables. The subgraph with root node $\mathbf{u}$ should be a POG representation of the formula $\psi|_\rho$.

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

1. If $\mathbf{u}$ is a literal ℓ', then the formula $\psi|_\rho$ must consist of the single unit clause $C = \{\ell'\}$, such that any $C' \in \mathrm{Prov}_\rho(C, \psi)$ 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 decision-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 $\mathbf{u}$, requiring at most two RUP steps.

3. If $\mathbf{u}$ is a product node, then we can divide its children into a set of literal nodes λ and a set of nonterminal nodes $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k$.

   (a) For each literal $\ell \in \lambda$, we must prove that any total assignment α satisfying ψ and such that $\rho \subseteq \alpha$ 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. (The solver is constrained to only use RUP inference rules, preventing it from introducing extension variables.)

   (b) For a single nonterminal child ($k = 1$), we recursively call $\mathsf{validate}(\mathbf{u}_1, \rho \cup \lambda, \psi)$.

   (c) For multiple nonterminal children ($k > 1$), it must be the case that the clauses in $\gamma = \psi|_{\rho \cup \lambda}$ 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 $\mathsf{validate}(\mathbf{u}_i, \rho \cup \lambda, \psi_i)$.

   We then generate the target clause for node $\mathbf{u}$ with a single RUP step, creating the hint by combining the results from the BCP and SAT calls for the literals, the recursively computed target clauses, and all but the first defining clause for node $\mathbf{u}$ (see Table 2(A)).

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

As examples, the forward implication proof of Figure 2(E) was generated by the structural approach. Working from step 36 backward, we can see that steps 35 and 36 complete the call to $\mathsf{validate}(\mathbf{s}_{10}, \emptyset, \phi_I)$. This call used x_1 as the splitting variable, first calling $\mathsf{validate}(\mathbf{p}_8, \{\overline{x}_1\}, \emptyset, \phi_I)$, which completed with step 29, and $\mathsf{validate}(\mathbf{p}_9, \{x_1\}, \phi_I)$, which completed with step 34. We see that each of these calls required separate traversals of nodes $\mathbf{s}_7$, $\mathbf{p}_6$, and $\mathbf{p}_5$, with the former yielding proof steps 25–27 and the latter yielding proof steps 30–32. This demonstrates how our simple formulation of $\mathsf{validate}$ effectively expands the graph into a tree. This shortcoming is avoided by the use of lemmas, as is described in Section 9.2.

8.2 Reverse Implication Proof

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

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

Each input clause deletion can be done as a single RUP step, based on an algorithm to test for clausal entailment in d-DNNF graphs (Darwiche & Marquis, 2002; Capelli, 2019). 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 does not satisfy C will extend to assignment α^* such that $\alpha^*(r) = 0$. 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 forces $\alpha^*(u) = 0$. 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 the construction of 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.

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 (Section 12.5) show that these optimizations yield substantial benefits.

9.1 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 $\ell \in \lambda$ 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 α consistent with the partial assignment ρ, must, when extended to α^*, yield $\alpha^*(v) = 1$. 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.

9.2 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 (Darwiche, 2002). When this decision-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 $\gamma_{\mathbf{u}}$ must extend to assignment α^* such that $\alpha^*(u) = 1$. 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 a product node $\mathbf{v}$ with arguments $\overline{\ell}_1, \overline{\ell}_2, \ldots, \overline{\ell}_k$, 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 cause u to be assigned 1. More precisely, given assignment α and its extension α^*, if $\alpha^*(v_j) = 0$ for every guard literal $\overline{v}_j$, then $\alpha^*(u) = 1$.

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 $\mathsf{validate}$.

Observe that the lemma structure can be hierarchical, since a shared subgraph may contain nodes that are themselves roots of shared subgraphs. Even then, 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.

9.3 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, when extended, assign 1 to s_7. Clauses 35 and 36 state that an assignment with $\alpha(x_1) = 0$ will, when extended, 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.

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 (de Moura & Ulrich, 2021). Verification was greatly aided by the Aesop (Limperg & From, 2023) automated proof search tactic. Lean 4 is based on a logical foundation in which expressions have a computational interpretation. As in other proof assistants such as Isabelle (Nipkow, Paulson, & Wenzel, 2002) and Coq (The Coq development

(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

team, 2024), functions defined in the formal system can be compiled to machine code. At the same time, we can state and prove claims about them within the same system, thereby verifying that our functions compute the intended results. In this section, we describe the functionality we implemented, what we proved about it, and the assumptions we made.

Data structures and mathematical model. When thinking about formal verification, it is helpful to distinguish between data structures that play a role in the code being executed, and *ghost* definitions that serve as a mathematical model, allowing us to state and prove specifications, but are erased during compilation and not executed. In the codebase, we generally store definitions in the two classes under `Data/` and `Model/`, respectively.

Among the former is our representation of CNF formulas. Following the DIMACS CNF convention, a variable is represented as a positive natural number, a literal is a non-zero integer, a clause is an array of literals, and a CNF formula is an array of clauses.

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

A POG is represented as a flat array of elements. Each element `PogElt` of a POG is either a variable, a binary disjunction (sum), or an arbitrary conjunction (product).

```
inductive PogElt where
  | var (x : Var) : PogElt
  | disj (x : Var) (l r : ILit) : PogElt
  | conj (x : Var) (args : Array ILit) : PogElt
```

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

On the mathematical side, our specifications rely on a general theory of propositional logic mirroring Section 3. The type `PropForm` describes the syntax of propositional formulas. It is generic over the type of variables, so we instantiate it with numeric variables as `PropForm Var`.

```
inductive PropForm (ν : Type u)
  | var (x : ν)
  | tr
  | fls
  | neg    (φ : PropForm ν)
  | conj   (φ₁ φ₂ : PropForm ν)
  | disj   (φ₁ φ₂ : PropForm ν)
  | impl   (φ₁ φ₂ : PropForm ν)
  | biImpl (φ₁ φ₂ : PropForm ν)
```

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`.

Functions `ILit.toPropForm`, `IClause.toPropForm`, `ICnf.toPropForm`, and `Pog.toPropForm` relate data structures to the formulas they encode. For example, given a literal u, `P.toPropForm` u denotes the interpretation of the node **u** corresponding to u in the POG P as a propositional formula $\phi_{\mathbf{u}}/\neg\phi_{\mathbf{u}}$ over the input variables. It is negated if u has negative polarity. Lean provides a convenient "anonymous projection" notation that allows writing `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 composite formulas, we found it easier to work with propositional formulas modulo logical equivalence, a structure known in logic as the *Lindenbaum–Tarski algebra*, rather than using `PropForm` directly. Its advantage is that equivalent but not syntactically equal formulas (such as $x \vee \neg x$ and $\top$) give rise to equal elements in the algebra, and equality has a privileged position in proof assistants based on type theory: equals can be substituted for equals in any context. In this way, forgetting syntactic detail is helpful. On the other hand, using the algebra gives rise to some challenges. The algebra, called `PropFun`, is defined as a quotient, with Boolean operations and the entailment relation lifted from the syntax of formulas to the new type. It is no longer straightforward to say when an element of the quotient "depends" on a variable since equivalent formulas can refer to different sets of variables. Instead, we use a semantic notion of dependence in which an element ϕ of the quotient depends on a variable x if and only if there is a truth assignment that satisfies ϕ, but falsifies ϕ after x is flipped.

```
/-- The semantic variables of 'φ' are those it is sensitive to as a Boolean
function. Unlike 'vars', this set is stable under equivalence of formulas. -/
def semVars (φ : PropFun ν) : Set ν :=
  { x | ∃ (τ : PropAssignment ν), τ ⊨ φ ∧ τ.set x (!τ x) ⊭ φ }
```

Proof checking. The goal of a CPOG proof is to construct a POG that is equivalent to the input CNF ϕ_I. The database of active clauses, the POG being constructed, and its root literal, are stored in a checker state structure `PreState`. The checker begins by parsing the input formula, initializing the active clauses to $\theta \leftarrow \phi_I$, and initializing the POG P to an empty one. It then processes every step of the CPOG proof, either modifying its state by adding/deleting clauses in θ and adding nodes to P, or throwing an exception if a step is incorrect. Afterwards, it carries out the FINAL CONDITIONS check of Section 7.2.

Throughout the process, we maintain invariants needed to establish the final result. These ensure that P is partitioned and that a successful final check entails the logical equivalence of ϕ_I and $\phi_{\mathbf{r}}$, where $\mathbf{r}$ is the final POG root (Theorem 1). Formally, we define a type `State` consisting of those `PreState`s that satisfy all the invariants. A `State` is a structure combining `PreState` fields with additional ones storing computationally irrelevant *ghost state* that asserts the invariants. The fields of st : `PreState` include `st.inputCnf` for ϕ_I, `st.clauseDb` for θ, and `st.pog` for P. We write `st.pogDefsForm` for the clausal 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 invariant states that assignments to input 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
```

Note that in the definition of uniqueExt, the arrows associate to the right, so the definition says that the three assumptions imply the conclusion. The next invariant guarantees that the set of solutions over the input 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)
```

The bulk of our work involved showing that these invariants are indeed maintained by the checker when going through a valid CPOG proof, modifying the active clause database and the POG. Together with additional, technical invariants about the correctness of cached computations, they imply the soundness theorem for P with root node $\mathbf{r}$:

Theorem 1. *If the proof checker has assembled POG P with root node $\mathbf{r}$ starting from input formula ϕ_I, and* FINAL CONDITIONS *(as stated in Section 7.2) hold of the checker state, then ϕ_I is logically equivalent to $\phi_\mathbf{r}$.*

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

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

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

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

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

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

Proposition 1 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 appearing in φ, we have that the number of models of φ over its variables is `countModels φ (card (vars φ))`.

Trust. To conclude this section, let us 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 (Ullrich & de Moura, 2019). Finally, Lean's standard library provides hashmap and array data structures. Most of their basic properties have been formally verified, but not all. We make use of some hashmap and array operations whose verification is not complete. Thus our proofs depend on the assumption that these operations have 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 implementations of basic data structures satisfy their descriptions, 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.

11. Implementation

We have implemented programs that, along with the D4 knowledge compiler, form the toolchain illustrated in Figure 1.[4] The proof generator is the same in both cases, since it need not be trusted. Our *verified* version of the proof checker and ring evaluator have been formally verified within the Lean 4 theorem prover. Our long term goal is to rely on these. Our *prototype* version is written in C. It is faster and more scalable, but we anticipate its need will diminish as the verified version is further optimized.

Our proof generator is written in C/C++ and uses a recent version of the CADICAL SAT solver that directly generates hinted proofs in LRAT format (Pollitt, Fleury, & Biere, 2023). It also uses their tool LRAT-TRIM to reduce the length of the generated proofs.

Section 7.5 presented two methods for generating the forward implication proof: a monolithic method relying on a single call to a proof-generating SAT solver, and a structural method that traverses the POG recursively and generates proof assertions for each node encountered. We devised an approach that combines the two, forming our *hybrid* method. Based on problem parameters, this approach starts with a top-down recursion, as with the structural method, but it shifts to a monolithic method once the subgraph size drops below a threshold. Section 12.2 describes the experiments used to determine the parameters for this approach in more detail.

---

4. The source code for all tools, as well as the Lean 4 derivation and checker, is available at `https://zenodo.org/records/14933727`.

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

We incorporated a ring evaluator into the prototype checker. It can perform both unweighted and weighted model counting with full precision. 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 unweighted 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. Our implementation of arbitrary-range integers represents a number as a sequence of "digits" with each digit ranging from 0 to $10^9 - 1$, and with the digits stored as four-byte blocks. This allows easy conversion to and from a decimal representation of the number.

12. Experimental Evaluation

Our experimental results seek to answer the following questions:

- How can a hybrid approach for the forward implication proof generation take advantage of the relative strengths of the monolithic and structural approaches?

- How well does our toolchain perform on actual benchmark problems?

- How strongly does our toolchain rely on the structure of the POG?

- How effective are the optimizations presented in Section 9?

- How does the verified proof checker perform, relative to the prototype checker?

- How does our toolchain perform compared to other tools for verifying the results of knowledge compilation and model counting?

12.1 Methodology

All experiments were run on a 2021 Apple MacBook Pro, with a 3.2 Ghz Apple M1 processor and 64 GB of RAM. We used a Samsung T7 solid-state disk (SDD) for file storage. We found that using an SSD was critical for dealing with the very large proof files (some over 150 GB).

As described in Section 5, we define the size of POG P to be to be the the number of nonterminal nodes plus the number of edges from these nodes to their children. This is also equal to the total number of defining clauses for the POG sum and product operations.

For benchmark problems, we used the public problems from the 2022 unweighted and weighted model counting competitions.[5] We found that there were 180 unique CNF files

---

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

among these, ranging in size from 250 to 2,753,207 clauses. With a runtime limit of 4,000 seconds, D4 completed for 123 of the benchmark problems. Our proof generator was able to convert all but one of these into POGs, with their declarations ranging from 304 to 1,765,743,261 (median 774,883) defining clauses. The additional problem would require 2,761,457,765 defining clauses, and this count overflowed the 32-bit signed integer we use to represent clause identifiers.

To make some of the experiments more tractable, we also created a reduced benchmark set, consisting of 90 out of the 123 problems for which D4 ran in at most 1000 seconds, and the generated POG had at most 10^7 defining clauses. These ranged in size from 304 to 8,493,275 defining clauses, with a median of 378,325.

Over the course of our tool development and evaluation, we have run D4 thousands of times. Significantly, we have not encountered any case where D4 generated an incorrect result.

We found that computing the *tree ratio* of a POG provides a useful metric for the degree of sharing among subgraphs. Formally, define the *tree size* of node $\mathbf{u}$, denoted $T(\mathbf{u})$, recursively:

- When $\mathbf{u}$ is a terminal node, $T(\mathbf{u}) = 0$.

- When $\mathbf{u}$ is a nonterminal node, with children $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k$:

$$T(\mathbf{u}) \quad = \quad k + 1 + \sum_{1 \le i \le k} T(\mathbf{u}_i) \tag{5}$$

A POG P with root node $\mathbf{r}$ is then defined to have a tree ratio $T(\mathbf{r})/|P|$. The tree size of a POG measures its size if all shared subgraph were expanded such that the graph is transformed into a tree. The tree ratio then measures the extent of subgraph sharing. The 122 problems for which POGs were generated had tree ratios ranging between 1.0 and 52,410, with a median of 11.6. Considering that the tree size can be exponentially larger than the POG size, these ratios are fairly modest.

12.2 Designing a Hybrid Forward-Implication Proof Generator

Our first set of experiments applies full monolithic and full structural generation to the reduced benchmark set. Figure 4 shows a plot comparing the two approaches. Each axis shows the number of seconds to generate the forward implication proof for the POG, with the X axis indicating the monolithic approach and the Y axis indicating the structural approach. Data points to the left of the diagonal line ran faster with the monolithic method, while those to the right ran faster with the structural method. The data are divided into those having tree ratios below 5.0 and those having tree ratios above 5.0. Of the 90 problems, 38 are below this tree ratio, and 52 are above. As can be seen there is some correlation between the relative performance of the two approaches and the tree ratio. For the 90 problems:

- For those with tree ratios below 5.0, 26 ran faster with monolithic generation, 11 with structural, and 1 tied.

- For those with tree ratios above 5.0, 12 ran faster with monolithic generation, 39 with structural, and 1 tied.
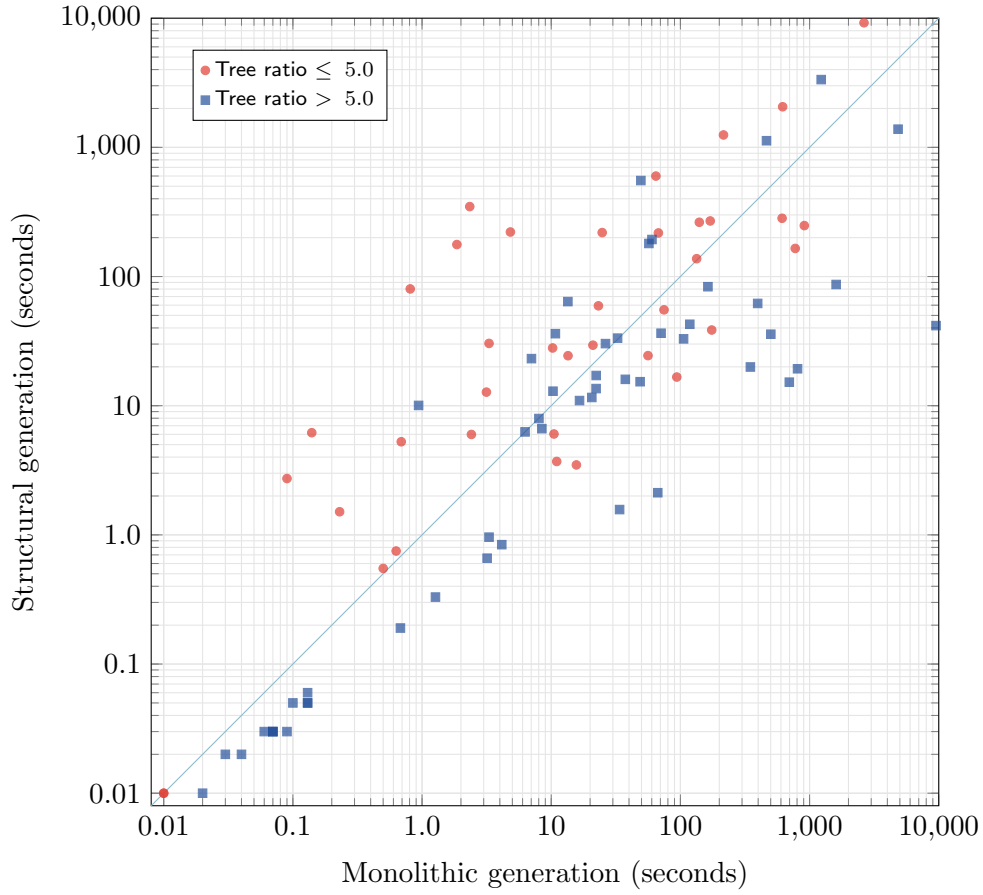
Figure 4: Structural (Y axis) versus monolithic (X axis) forward implication proof genera-
tion times. The structural approach generally performed better for formulas with
high tree ratios.

Figure 5 shows the comparative proof sizes (in clauses) for the two approaches. As can
be seen, the monolithic approach tends to generate shorter proofs. For the 90 problems,
72 had smaller proofs with monolithic generation and 18 with structural. There is little
correlation between the relative proof sizes and the tree ratios.

Based on the results for the reduced benchmark set, we devised the following selection
rule: when the tree ratio for the POG is at most 5.0, use the monolithic approach, otherwise
use the structural approach. That would yield the better choice, in terms of runtime, for
65 of the 90 cases. Our data set was too sparse to do more tuning, including a more refined
threshold selection.

We tried a variety of hybrid approaches, where the proof generator starts at the top
using a structural approach and then switches to a monolithic approach once the tree size
for a node drops below some threshold. This was helpful for very large problems, but setting
a low threshold (tree size less than 10^6) consistently led to poorer runtime performance.
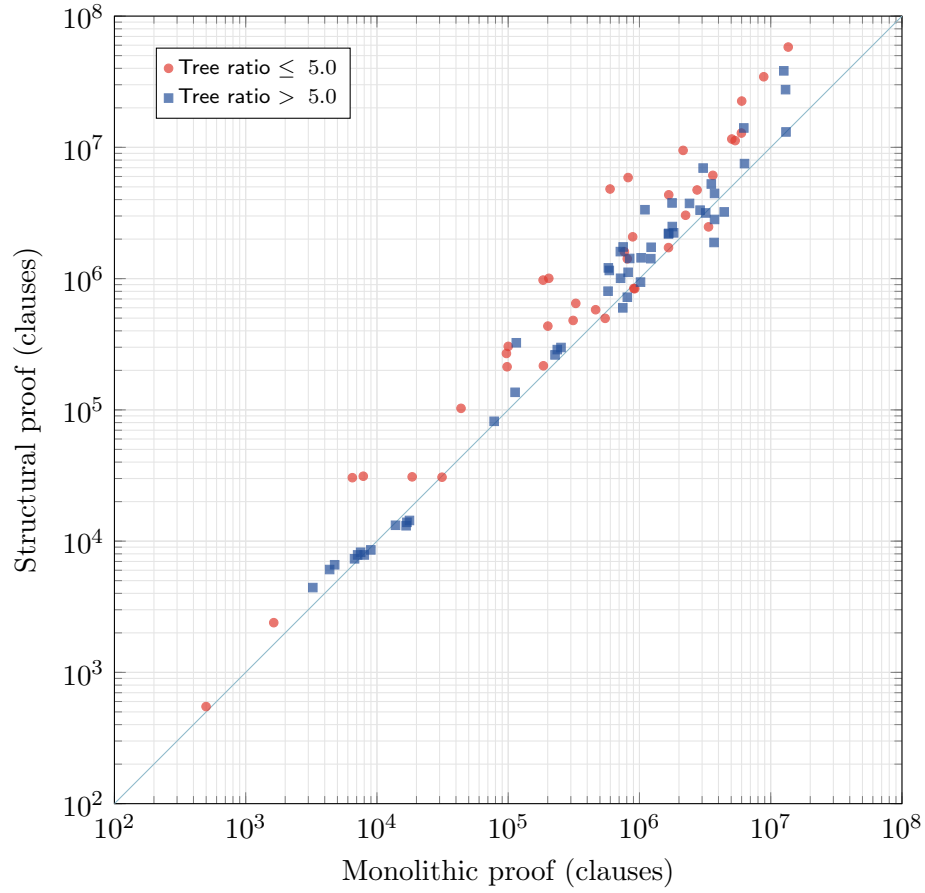
Figure 5: Structural (Y axis) versus monolithic (X axis) proof sizes. The monolithic approach generated shorter proofs in most cases.

We also found that the SAT solver could not reliably handle problems with more than 10^7 clauses. We therefore refined the rule for a hybrid approach that operates as follows:

1. With a bottom-up traversal of the graph, label each node by its tree size.

2. Compute the total size of the graph and the tree ratio of the root.

3. Proceed with proof generation with the following rules

   (a) If the tree ratio is at most 5.0, and the tree size of the root is below 10^6, do the entire proof generation with a monolithic approach

   (b) If the tree ratio is at most 5.0, and the tree size of the root is above 10^6, start with a structural approach and shift to a monolithic approach once the tree size at a node is below 10^6.

   (c) If the tree ratio is above 5.0, do the entire proof generation with a structural approach.
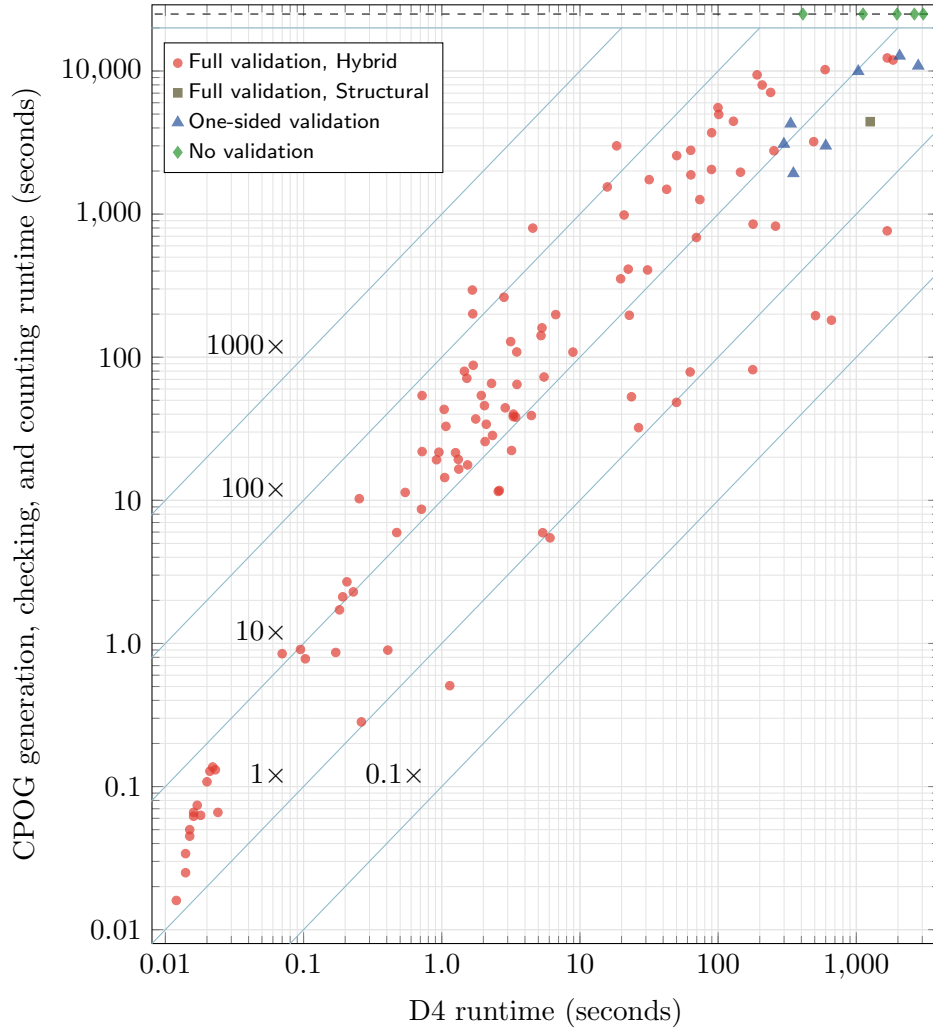
Figure 6: Combined runtime for CPOG proof generation, checking, and counting as function of D4 runtime. Timeouts are shown as points on the dashed line. Full verification completed for 111 of the 123 benchmark problems. The median ratio between the two times for the completed problems was 12.5.

Unless noted otherwise, the remainder of our experimental data is based on this approach.

12.3 Toolchain Performance Evaluation

Figure 6 shows the performance of our toolchain for the 123 problems for which D4 completed within 4,000 seconds. This figure shows the runtime for D4 on the X axis and the runtime for the toolchain on the Y axis. The toolchain included proof generation, proof checking with the prototype checker, and counting computation. The counting computation included unweighted model counting for each problem, plus weighted model counting for

those from the weighted model counting competition. We allowed a maximum of 10,000 seconds for the toolchain. For those problems that failed to complete within the time limit, we attempted other approaches. For those with low tree ratios, we attempted using a full structural approach. For those where we could not obtain a complete proof, we attempted a one-sided proof, generating only the reverse implication proof. The results can be summarized as follows:

- Of the 123 problems, 110 were completed using the hybrid approach.

- One additional problem completed with the structural approach (as well as by using a hybrid approach with the tree size limit set to 10^5.)

- For seven others, we were able to generate and check a one-sided proof.

- For five problems, no form of validation succeeded. This included the one for which the POG was too large to encode the clause identifiers.

As one might expect, the largest problems proved to be the most challenging. Of the four with more than 10^9 defining clauses, one completed with a one-sided proof, while the other three had no form of validation.

Figure 6 also allows comparing the time to validate the output of the knowledge compiler relative to the time for the compiler itself. (The counting computations had negligible impact on the overall toolchain performance.) For the 111 problems for which full proofs were generated and checked, the ratio between these two times ranged between 0.27 (i.e., validation was 3.64× faster than generation) and 177.0, with a median of 12.5. The ones with very high ratios tended to be ones with very few models, and so most of the proof generation time was spent generating unsatisfiability proofs.

It is encouraging that we could validate the results of a knowledge compiler for all but the largest problems. Nonetheless, the high ratio between our toolchain time and the time required by the compiler indicates that validation comes at a significant cost. By contrast, modern SAT solvers incur only a small performance penalty when generating proofs of unsatisfiability (Heule, Hunt Jr., & Wetzler, 2013a). With the advent of solvers that also generate hints for the proof steps (Pollitt et al., 2023), the proof checking overhead has also become very small.

Figure 7 compares the total number of clauses in the CPOG representation (Y axis) versus the number of defining clauses (X axis). Since the former include the latter, the ratio between these cannot be less than 1.0. The ratios ranged between 1.02× and 9460.2×. Again, the largest ratios were for problems with very few models, and hence most of the steps were for the unsatisfiability proofs in the literal justifications. The median ratio was 2.29×. This is a relatively modest overhead, although it requires transforming the large decision-DNNF files into even larger CPOG files.

12.4 Toolchain Robustness Evaluation

Although the CPOG framework is very general and makes no assumptions about how the the POG relates to the input CNF formula, our proof generator is less general. It requires that the POG arise from a decision-DNNF graph. Moreover, our structural approach requires
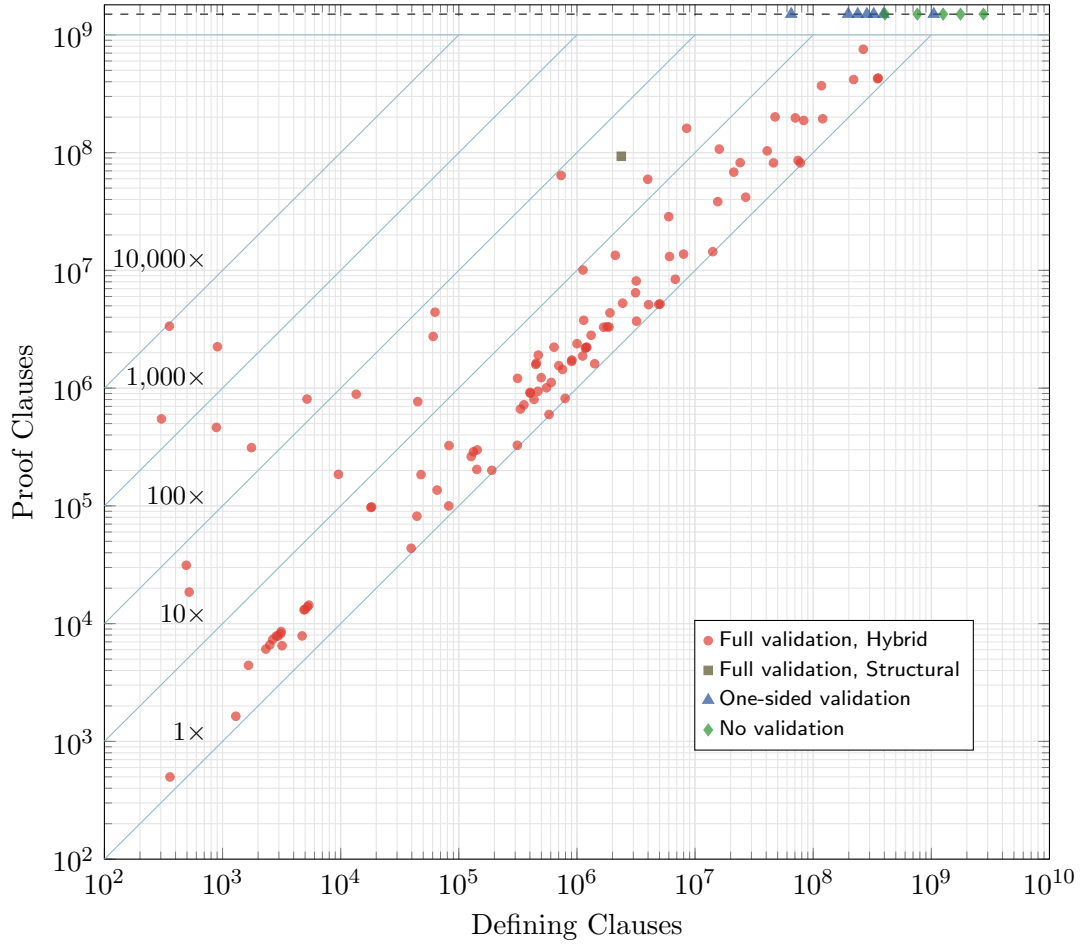
Figure 7: Total number of clauses in CPOG file as function of number of defining clauses. The median ratio of the two was 2.29.

that the CNF formula decompose according to the decision-DNNF structure. That is, as it recurses downward, the simplified clauses must be encoded by the POG subgraphs.

Our monolithic approach, on the other hand, makes no assumption about the relation between the POG and the CNF formula. As long as every satisfying assignment to the CNF would, when extended, cause the POG root to evaluate to 1, the monolithic approach can, in principle, generate a forward implication proof. Our reverse implication proof generation is also independent of any structural relations between the two representations.

We tested this hypothesis by using the preprocessing capabilities of D4 to transform the input formula into a different, but logically equivalent clausal representation. D4 can optionally perform three different forms of preprocessing (Lagniez & Marquis, 2021). These are designed to make knowledge compilation more efficient, but they also have the effect of creating a mismatch between the structure of the generated decision-DNNF graph and the original input formula.

We used the 90 problems from the reduced benchmark set as test cases, running D4 by preprocessing with all three methods enabled (these are referred to as "backbone," "vivification," and "occElimination") followed by knowledge compilation. None of the resulting POGs could be verified using the structural approach. Setting an overall time limit of 1000 seconds for the combination of D4 (including preprocessing), proof generation, proof checking, and counting, and using monolithic proof generation, we obtained the following results:

- For 7 problems, neither approach completed within 1000 seconds.

- For 2 problems, running with preprocessing completed within the time limit, while running without did not.

- For 1 problem, running without preprocessing completed within the time limit, while running with did not.

- For 45 problems, both completed, with the preprocessing version running faster.

- for 35 problems, both completed, with the preprocessing version running slower.

These results indicate that the preprocessing is only marginally effective. Importantly, however, they demonstrate that our toolchain can establish the end-to-end correctness of preprocessing plus knowledge compilation.

Even with monolithic mode, our proof generator still requires that the output of the knowledge compiler be a decision-DNNF graph. We discuss how it could be generalized even further in Section 13.1.

12.5 Effect of Optimizations

Section 9 describes two optimizations for proof generation: literal grouping and lemmas. These optimizations are only applied when using a structural approach, and so we focus our evaluation on the 52 problems having tree ratios greater than 5.0 from the reduced benchmark set of 90 problems.

Figure 8 summarizes the sizes of the CPOG representations generated for these problems with and without the optimizations. The X axis shows the size (in clauses) for the proof when neither optimization is enabled, while the Y axis shows the sizes with either one or both enabled. The extent to which a point lies below the diagonal line labeled "1×" therefore indicates the benefit of the optimizations. Two benchmarks required lemmas to complete. These are indicated along the far edge of the X axis. In the remaining, we consider mostly the 50 benchmarks for which all four variants completed.

Literal grouping alone (the hollow diamonds clustered along the diagonal line) has only minimal benefit. Compared to the unoptimized proof sizes, literal grouping yielded proofs that ranged between being $1.10\times$ larger and $1.10\times$ smaller, with a median ratio of 1.0. Although literal grouping reduces the number of unsatisfiability proofs that must be generated, the resulting proofs are enough larger to offset this advantage.

Using lemmas alone (the hollow pentagons), on the other hand, shows significant benefit. The resulting proofs were between $1.06\times$ and $52.54\times$ smaller, with a median of $7.95\times$. In addition, lemmas enable two benchmarks to complete that otherwise fail. These problems
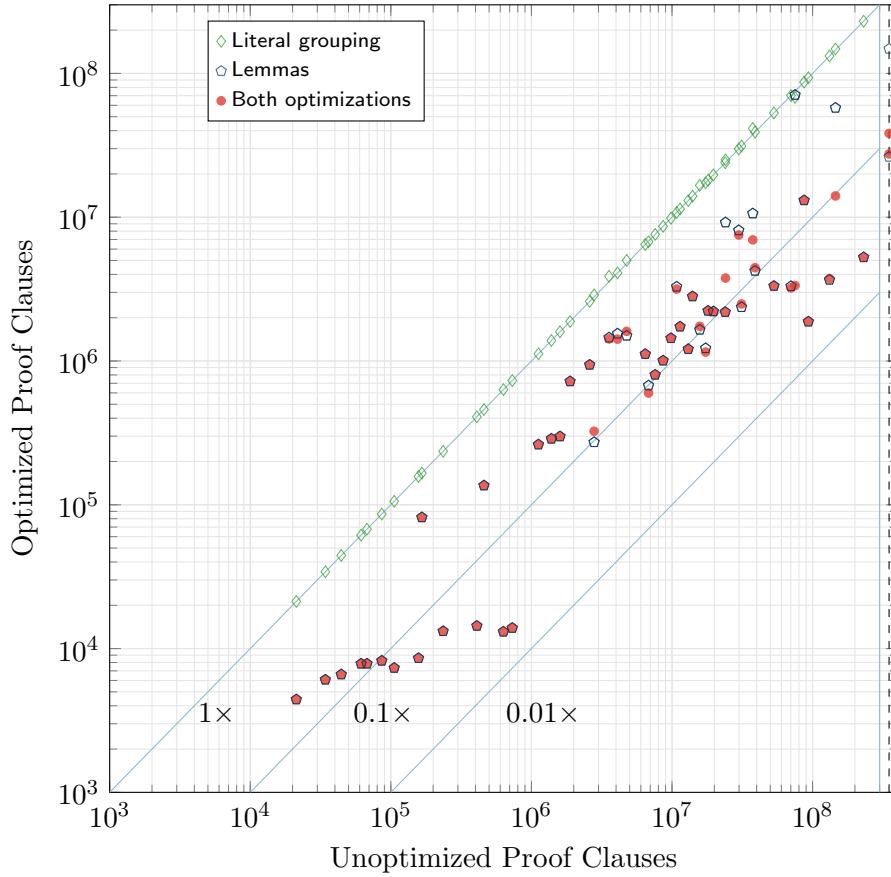
Figure 8: Proof clauses when one or both optimizations is enabled, versus without optimization. Lemmas provide substantial benefit, while the results for literal grouping are mixed.

have high degrees of subgraph sharing, and so the ability to avoid expanding the proofs into tree structures was important.

Combining literal grouping with lemmas (the solid dots) showed a modest improvement over using lemmas alone. Many of the solid dots coincide with or are very close to the hollow pentagons, with some being slightly better and others being slightly worse. Significantly, however, several problems showed major benefit from combining the two optimizations. In the most extreme case, one problem had between 68 and 75 million proof clauses with either no or a single optimization, but just 3.3 million with both optimizations. Compared to the unoptimized proofs, the combination yielded proofs ranging from $2.03\times$ to $52.54\times$ smaller, with a median of $8.59\times$.

The runtime improvement with the optimizations was smaller than the size improvement, but still significant. Generating shorter proofs enables the checker to run faster, and so there is some benefit in spending more time in proof generation to reduce the proof
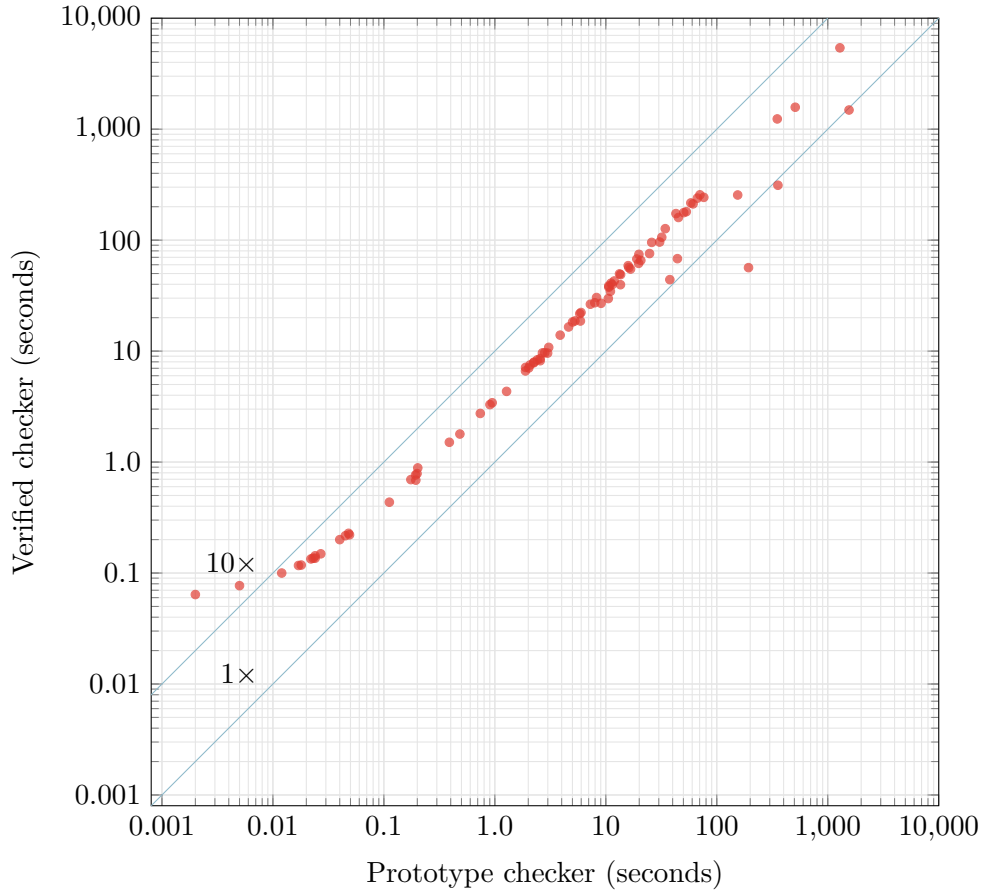
Figure 9: Times for Verified Checker versus Prototype Checker. Both show similar scaling.

size. We therefore consider the combined time to generate and to check the proofs. Literal grouping, on its own, caused the toolchain to run with a range from $4.02\times$ slower to $1.24\times$ faster, with a median slowdown of $1.70\times$, compared to no optimization. Lemmas, on their own, yielded speedups ranging from $1.03\times$ to $18.04\times$, with a median of $2.78\times$. Combining the two yielded performances ranging from a slowdown of $1.23\times$ to a speedup of $23.01\times$, with a median speedup of $3.02\times$.

Overall, these results indicate that lemmas provide an important optimization, while literal grouping provides a modest benefit.

12.6 Performance of the Formally Verified Proof Checker

Our prototype proof checker is fairly simple and has shown itself to be reliable, but we have not subjected it to rigorous, adversarial testing. Using our verified checker removes any doubt about the trustworthiness of the compiled result. For the 90 problems from the reduced set, we generated CPOG files using the hybrid approach and ran both checkers.
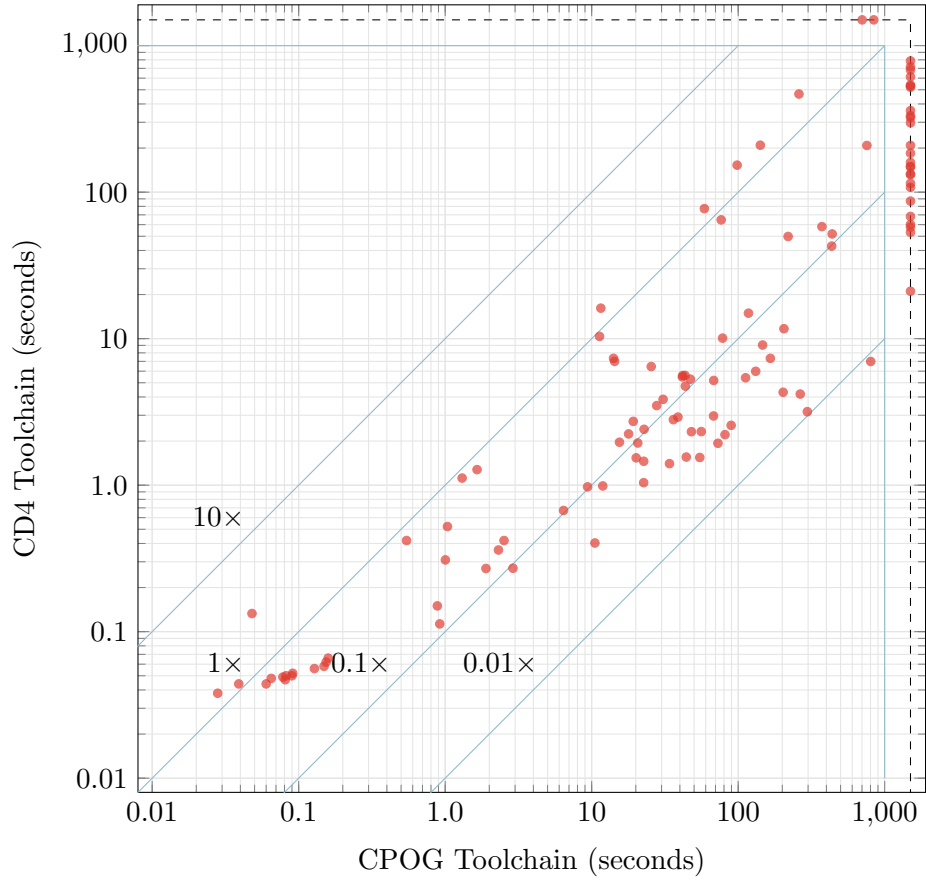
Figure 10: Times for CD4 Toolchain versus CPOG Toolchain. Times include knowledge compilation, proof generation, and checking.

Figure 9 summarizes the results, with the runtime for the prototype checker on the X axis and for the verified checker on the Y axis.

We can see in this figure that the verified checker has a startup time of around 70 milliseconds, causing it to run much slower compared to the prototype checker on the very small problems. If we consider only the 76 problems requiring more than 0.1 seconds with the prototype checker, we see that the verified checker runs between 3.42× faster and 4.39× slower than the prototype, with a median of 3.54× slower.

Significantly, the relative performance remains constant even for the larger proofs, showing that the two programs have similar scaling properties.

12.7 Comparison to Other Validation Frameworks

As described in Section 2, two other verification frameworks have been developed that are relevant to ours: the CD4 framework (Capelli, 2019; Capelli et al., 2021), designed for the D4 knowledge compiler, and the MICE framework (Fichte et al., 2022), designed to verify

unweighted model counters. Here we compare how they perform on all 123 problems in the full benchmark set.

Running CD4 involves running D4 with appropriate arguments.[6] Checking the results requires running two checkers: one for the annotated decision-DNNF graph, plus DRAT-TRIM for the generated proof clauses. The first checker is not available in any public repository. We used a copy supplied to us by the authors. The combined toolchain therefore involves running the knowledge compiler and the two proof checkers. For comparison, we consider the time for our complete toolchain, including running D4, the proof generator, and the prototype proof checker. For both toolchains, we set a time limit of 1,000 seconds. We ran both toolchains for all 123 problems.

Figure 10 compares times for the toolchains, with those for our toolchain on the X axis and those for the CD4 toolchain on the Y axis. The results can be summarized as follows:

- Both toolchains completed for 82 problems, with 8 running faster with our toolchain and 74 running faster with the CD4 toolchain. Overall, our toolchain ranged from $2.77\times$ faster to $114.91\times$ slower, with a median of running $7.81\times$ slower.

- Our toolchain completed 2 problems for which the CD4 toolchain did not complete within 1000 seconds.

- The CD4 toolchain completed 26 problems for which our toolchain did not complete within 1000 seconds.

- Neither toolchain completed for 13 problems.

Clearly, CD4 has better overall scaling and performance. Even with a time limit of 1000 seconds, it was able to handle all but 15 of the 123 problems.

The CD4 toolchain has impressive performance, but as a general tool it has significant shortcomings. It relies strongly on the inner workings of the knowledge compiler. It cannot even verify its own output when preprocessing is enabled. Furthermore, even having corrected the known flaw, there is no guarantee that their framework is sound or that their checker is correct.

Running MICE on the output of a knowledge compiler requires running two programs: NNF2TRACE, a proof generator for decision-DNNF graphs, and SHARPTRACE, a checker for the generated proofs.[7]

The results for the reduced set of 90 problems are shown in Figure 11, comparing the time to generate and check the proofs with our framework on the X axis, and the time to do so with the MICE tools on the Y axis. Both were set to have a time limit of 1000 seconds. The results can be summarized as follows:

- Both toolchains completed for 75 problems, with 66 running faster with our toolchain and 9 running faster with the MICE toolchain. Overall, our toolchain ranged from $3461\times$ faster to $368\times$ slower, with a median of running $7.67\times$ faster.

---

6. This is possible with the original version of D4, available at `https://github.com/crillab/d4`. It was not incorporated into the more recent version, available at `https://github.com/crillab/d4v2`.
7. Both programs were downloaded from `https://github.com/vroland`.
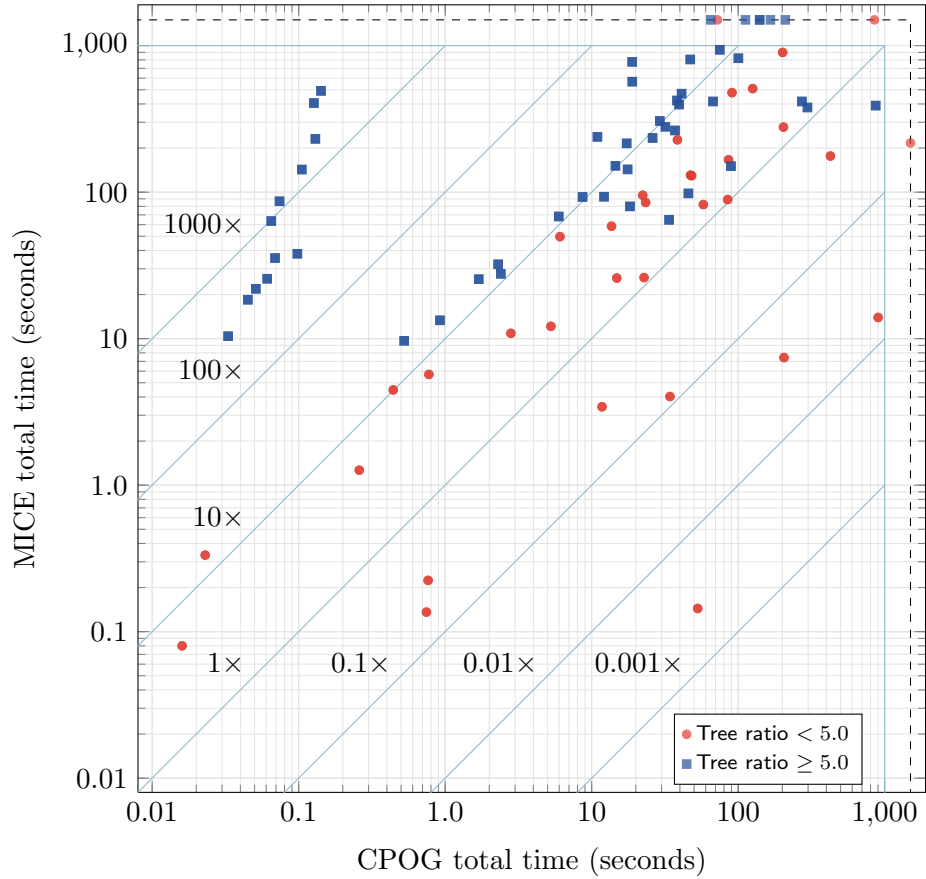
Figure 11: Running Time for MICE versus our proof chains. Times include proof genera-
tion, checking, and counting. Timeouts are shown as points on the dashed lines.
MICE is especially weak on problems with high tree ratios.

- Our toolchain completed 7 problems for which the MICE toolchain did not complete within 1000 seconds.

- The MICE toolchain completed 1 problem for which our toolchain did not complete within 1000 seconds.

- Neither toolchain completed for 7 problems.

One shortcoming of the MICE framework is highlighted by the division of the data points in Figure 11 according to tree ratios. Those with tree ratios above 5.0 consistently performed poorly for MICE, with 5 exceeding the time limit and 43 requiring more time than with our toolchain. Only 1 problem above this threshold ran faster with MICE than with ours. These are the problems with significant amounts of sharing in the subgraph. Our toolchain exploits this sharing by generating and using lemmas for the shared subgraphs. MICE, on the other hand, has no mechanism for reusing results, effectively expanding the graphs into trees.

Overall, these results indicate that the MICE framework has serious performance limitations, due in part to its inability to efficiently exploit the sharing of subgraphs. In addition, the MICE proof generator relies strongly on the means by which the knowledge compiler output was generated. For example, it cannot perform an end-to-end verification of the combination of preprocessing and knowledge compilation. Other shortcomings include the lack of formal verification for the framework or the checker, and that the framework can only validate the unweighted model count.

13. Extensions

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 current implementation only handles the outputs of the D4 knowledge compiler, and it supports only queries that can be computed via ring evaluation. Here we discuss ways to extend both capabilities.

13.1 Validating Arbitrary POGs

Extending our proof generator to other knowledge compilers that generate decision-DNNF, such as DSHARP (Muise, McIlraith, & Beck, 2012), requires simply extending the parser. Some knowledge compilers, however, generate representations that cannot be directly encoded into decision-DNNF. For example, the Sentential Decision Diagram representation introduced by Darwiche (2011) can readily be translated into d-DNNF, but with the possibility that some sum nodes will not have associated decision variables.

Extending our tool to handle arbitrary POGs, including d-DNNF as a subset, could be done with modest effort. Our monolithic approach can generate forward implication proofs for this more general form. Our method for generating reverse implication proofs currently handles d-DNNF formulas (Darwiche & Marquis, 2002; Capelli, 2019), but not formulas with negations. Extending it to POGs would require marking nodes for both negative and positive polarities. The proof generator must also generate mutual exclusion generate proofs for each sum node declaration. This could be done with a proof-generating SAT solver. That is, for child nodes $\mathbf{u}_0$ and $\mathbf{u}_1$, it would generate a CNF formula θ_c consisting of the defining clauses for the subgraphs having $\mathbf{u}_0$ and $\mathbf{u}_1$ as roots, and run a SAT solver on $\theta_c|_{\{u_0,u_1\}}$, the formula that would be satisfied by an extended assignment α^* that assigns value 1 to both children. The proof of unsatisfiability can then be translated into a series of clause additions, adding literals $\overline{u}_0$ and $\overline{u}_1$ to each proof clause. The hint for the final proof step then serves as the hint for the mutual exclusion proof in the sum declaration.

13.2 Generalizing to Semirings

The formulation of algebraic model counting by Kimmig et al. (2017) is more general than ours. It allows the algebraic structure to be a *semiring*. A commutative semiring $\mathcal{S}$ obeys all properties of a commutative ring, except that the elements of the set need not have additive inverses. We can define the *semiring evaluation problem* as computing

$$\mathbf{S}(\phi, w) \;=\; \sum_{\alpha \in \mathcal{M}(\phi)} \; \prod_{\ell \in \alpha} w(\ell) \tag{6}$$

where sum $\sum$ is computed according to the semiring addition operation $+$ and product $\prod$ is computed according to the semiring product operation $\times$.

As an example, consider the formulation of the weighted model counting computation in Section 4, but using max as the sum operation, rather than addition. The computation would then yield the maximum weight for all satisfying assignments, rather than their sum.

Semiring evaluation can be performed via knowledge compilation by requiring that the representation generated by the compiler be in negation normal form, and that it obey a property known as *smoothness* (Darwiche & Marquis, 2002; Shih, Van den Broeck, Beame, & Amarilli, 2019). Within our formulation, a partitioned-operation formula is smooth when all arguments to each sum operation have identical dependency sets. That is, every sum operation $\bigvee_{1 \leq i \leq k} \phi_i$ has $\mathcal{D}(\phi_i) = \mathcal{D}(\phi_1)$ for $1 < i \leq k$. Smoothness can be ensured by adding redundant formulas to artificially introduce variables. For example, if subformula ϕ_i lacks having variable x in its dependency set, it can be replaced by $(x \vee^{\mathsf{p}} \overline{x}) \wedge^{\mathsf{p}} \phi_i$. When a knowledge compiler generates a representation in negation normal form that is smooth, then a semiring evaluation of the formula can proceed by first assigning each literal ℓ the value $w(\ell)$. Then the product and sum operations are evaluated in manners analogous to (3) and (4).

Our POG representation can support evaluation of semiring formulas by imposing the restriction that the POG is in negation normal form and that it is smooth. Given a smoothed decision-DNNF graph generated by a knowledge compiler, our toolchain will convert this into a smooth POG in negation normal form and verify its equivalence to the input formula. Full verification would also require checking that the POG is smooth. We must also extend the formal derivation to ensure soundness and to create a formally verified checker.

14. Concluding Remarks

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

Our experiments demonstrate that our toolchain can already handle problems nearly at the limits of current knowledge compilers. Further engineering and optimization of our proof generator and checker could improve their performance and capacity substantially. We also show that, by using monolithic proof generation, our toolchain can be agnostic to the means by which the knowledge compiler created a decision-DNNF representation of the input formula. This generality, plus the fact that our toolchain has been formally verified, provides a major improvement over previous methods for checking the outputs of knowledge compilers and model counters.

Acknowledgments

References

Beame, P., Li, J., Roy, S., & Suciu, D. (2013). Lower bounds for exact model counting and applications in probabilistic databases. In *Uncertainty in Artificial Intelligence*, pp. 52–61.

Blum, M., Chandra, A. K., & Wegman, M. N. (18 March 1980). Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, *10*(2), 80–82.

Bryant, R. E., Nawrocki, W., Avigad, J., & Heule, M. J. H. (2023). Certified knowledge compilation with application to verified model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl.

Capelli, F. (2019). Knowledge compilation languages as proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*, Vol. 11628 of *LNCS*, pp. 90–91.

Capelli, F., Lagniez, J.-M., & Marquis, P. (2021). Certifying top-down decision-DNNF compilers. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Cruz-Filipe, L., Heule, M. J. H., Hunt, W. A., Kaufmann, M., & Schneider-Kamp, P. (2017). Efficient certified RAT verification. In *Conference on Automated Deduction (CADE)*, Vol. 10395 of *LNCS*, pp. 220–236.

Darwiche, A. (2001). Demposable negation normal form. *Journal of the ACM*, *48*.

Darwiche, A. (2002). A compiler for deterministic, decomposable negation normal form. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Darwiche, A. (2011). SDD: A new canonical representation of propositional knowledge bases. In *International Joint Conference on Artificial Intelligence*, pp. 819–826.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, *17*.

de Moura, L., & Ulrich, S. (2021). The Lean 4 theorem prover and programming language. In *Conference on Automated Deduction (CADE)*, Vol. 12699 of *LNAI*, pp. 625–635.

Fichte, J. K., Hecher, M., & Roland, V. (2022). Proofs for propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Goldberg, E. I., & Novikov, Y. (2003). Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE)*, pp. 886–891.

Heule, M. J. H., Hunt, W. A., Kaufmann, M., & Wetzler, N. D. (2017). Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving*, Vol. 10499 of *LNCS*, pp. 269–284.

Heule, M. J. H., Hunt Jr., W. A., & Wetzler, N. D. (2013a). Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 181–188.

Heule, M. J. H., Hunt, Jr., W. A., & Wetzler, N. D. (2013b). Verifying refutations with extended resolution. In *Conference on Automated Deduction (CADE)*, Vol. 7898 of *LNCS*, pp. 345–359.

Heule, M. J. H., Seidl, M., & Biere, A. (2014). A unified proof system for QBF preprocessing. In *International Joint Conference on Automated Reasoning (IJCAR)*, Vol. 8562 of *LNCS*, pp. 91–106.

Huang, J., & Darwiche, A. (2007). The language of search. *Journal of Artificial Intelligence Research*, *22*, 191–219.

Järvisalo, M., Heule, M. J. H., & Biere, A. (2012). Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, Vol. 7364 of *LNCS*, pp. 355–370.

Kimmig, A., den Broeck, G. V., & Raedt, L. D. (2017). Algebraic model counting. *Journal of Applied Logic*, *22*, 46–62.

Lagniez, J.-M., & Marquis, P. (2017). An improved decision-DNNF compiler. In *International Joint Conference on Artificial Intelligence*, pp. 667–673.

Lagniez, J.-M., & Marquis, P. (2021). Preprocessing for propositional model counting. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Lammich, P. (2020). Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, *64*(3), 513–532.

Limperg, J., & From, A. H. (2023). Aesop: White-box best-first proof search for Lean. In *Certified Programs and Proofs (CPP)*, pp. 253–266. ACM.

Monet, M., & Olteanu, D. (2018). Towards deterministic decomposable circuits for safe queries. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*.

Muise, C., McIlraith, S. A., & Beck, J. C. (2012). Dsharp: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*.

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL. A proof assistant for higher-order logic*, Vol. 2283 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin.

Pollitt, F., Fleury, M., & Biere, A. (2023). Faster LRAT checking than solving with CaDiCaL. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J.ACM*, *12*(1), 23–41.

Shih, A., Van den Broeck, G., Beame, P., & Amarilli, A. (2019). Smoothing structured decomposable circuits. In *Advances in Neural Information Processing Systems*, Vol. 32.

Tan, Y. K., Heule, M. J. H., & Myreen, M. O. (2021). cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*, Vol. 12652 of *LNCS*, pp. 223–241.

The Coq development team (2024). The Coq proof assistant.. https://coq.inria.fr/.

Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pp. 466–483. Springer.

Ullrich, S., & de Moura, L. (2019). Counting immutable beans: reference counting optimized for purely functional programming. In *Implementation and Application of Functional Languages (IFL)*, pp. 3:1–3:12. ACM.

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

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

Wetzler, N. D., Heule, M. J. H., & Hunt Jr., W. A. (2014). DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, Vol. 8561 of *LNCS*, pp. 422–429.

Zhang, L., & Malik, S. (2003). Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE)*, pp. 880–885.