Certifying Projected Knowledge Compilation

Randal E. Bryant
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Yong Kiam Tan
Institute for Infocomm Research (I^2R), A*STAR, Singapore
Nanyang Technological University, Singapore

Marijn J. H. Heule
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract

Knowledge compilers convert Boolean formulas, given in conjunctive normal form (CNF), into representations that enable efficient evaluation of unweighted and weighted model counts, as well as a variety of other useful properties. With *projected* knowledge compilation, the generated representation describes the restriction of the formula to a designated set of *data* variables, with the remaining ones eliminated by existential quantification. Projected knowledge compilation has applications in a variety of domains, including formal verification and synthesis.

This paper describes a formally verified proof framework for certifying the output of a projected knowledge compiler. It builds on an earlier clausal proof framework for certifying the output of a standard knowledge compiler. Extending the framework to projected compilation requires a method to represent Skolem assignments, describing how the quantified variables can be assigned, given an assignment for the data variables. We do so by extending the representation generated by the knowledge compiler to also encode Skolem assignments. We also refine the earlier framework, moving beyond purely clausal proofs to enable scaling certification to larger formulas.

We present experimental results obtained by making small modifications to the D4 projected knowledge compiler and extensions of our earlier proof generator. We detail a soundness argument stating that a compiler output that passes our certifier is logically equivalent to the quantified input formula; the soundness argument has been formally validated using the HOL4 proof assistant. The checker also ensures that the compiler output satisfies the properties required for efficient unweighted and weighted model counting. We have developed two proof checkers for the certification framework: one written in C and designed for high performance and one written in CakeML and formally verified in HOL4.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Knowledge Compilation, Propositional model counting, Proof checking

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.2

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15628075`

Funding *Randal E. Bryant*: National Science Foundation, NSF grant CCF-2108521
Yong Kiam Tan: Singapore NRF Fellowship Programme NRF-NRFF16-2024-0002
Marijn J. H. Heule: National Science Foundation, NSF grant CCF-2108521

1 Motivation: (Projected) Knowledge Compilation

Although Boolean satisfiability (SAT) solvers serve as effective tools for many automated reasoning tasks, some applications require evaluating more detailed properties about formulas than whether or not they are satisfiable. For example, *model counting* [26] determines the number of satisfying assignments to a formula, while *probabilistic inference* [11] determines the probability that a formula will evaluate to 1 (true), given individual probabilities that each variable is assigned value 1. Probabilistic inference is a special case of the more general

problem of *weighted model counting*. Generalizing to weight functions computed over algebraic semirings expands properties that can be evaluated even further [28].

Rather than implementing specialized programs for these different analysis tasks, a *knowledge compiler* [18, 34] converts a Boolean formula into a form for which the tasks can be performed in polynomial time, relative to the size of the representation. Most commonly, knowledge compilers generate a restricted class of Boolean formulas for which the analysis becomes tractable, such as the deterministic, decomposable, negation-normal form (d-DNNF, defined in Section 4) [18]. These formulas are encoded as directed acyclic graphs, enabling efficient sharing of common subformulas. We can therefore state the operation of a knowledge compiler as converting a Boolean formula over a set of variables X, denoted $\phi(X)$, into a logically equivalent formula $\psi(X)$, such that (weighted) model counting can be performed in time polynomial in the size of ψ.

With *projected* knowledge compilation, the variables X are partitioned into *data* variables D and *quantified* variables Y, and so the input formula has the form $\phi(D, Y)$. The task is to produce a formula $\psi(D)$ where the quantified variables have been eliminated via existential quantification. That is, the satisfying assignments of ψ consist of those assignments α to the data variables, such that for some assignment β to the quantified variables, their combination $\alpha \cdot \beta$ satisfies the input formula ϕ. This can be written as $\psi(D) \equiv \exists Y\, \phi(D, Y)$. This form of projection is also referred to as "forgetting" [18, 21, 33].

The need to eliminate quantified variables arises in many contexts. For example, auxiliary variables are commonly introduced when encoding general Boolean formulas into conjunctive normal form (CNF), via either a Tseitin [38] or a Plaisted-Greenbaum [37] encoding. Although the former preserves model counts, the latter does not, and both can complicate the task of weighted model counting. As a second example, when a hardware system is described as the composition of a set of components, Boolean variables are introduced to encode the connections between components [13]. Existentially quantifying these variables abstracts away the internal system details. As a third example, a state transition system can be described via a *transition relation* $T(S, S')$ defining the allowed combinations of current states S and next states S' [8]. For some predicate $P(S)$ describing the current states, the quantified formula $P'(S') \doteq \exists S\, [P(S) \wedge T(S, S')]$ describes the set of possible successor states.

Our work makes use of version 2 of the D4 knowledge compiler [30].[1] Given a CNF file that also includes a declaration of the data variables (referred to as "show" variables), as is required for the projected model counting competitions,[2] it will generate a d-DNNF representation of the set of data models.

2 Certification Framework and Tools

For many applications of automated reasoning, including formal verification and mathematics, it is essential that the tools be fully trustworthy. In our context, there should be some method of ensuring that the output $\psi(D)$ generated by a projected knowledge compiler is truly equivalent to $\exists Y\, \phi(D, Y)$. We have devised a proof framework for expressing such proofs and a set of tools both to generate and to check these proofs. Such a proof enables the equivalence to be established for individual executions of the compiler, even if the overall correctness of the compiler cannot be guaranteed. In prior work [6, 7], we devised a framework for

[1] Available at `https://github.com/crillab/d4v2`
[2] `https://mccompetition.org`

certifying the results of standard knowledge compilation. This paper extends this framework and associated tools to handle projected knowledge compilation.

For standard knowledge compilation, the task is to prove that $\phi(X) \equiv \psi(X)$, i.e., that the CNF and the d-DNNF representations are logically equivalent. In our framework, d-DNNF formulas are represented by directed acyclic graphs that we refer to as "partitioned-operation graphs" (POGs). These graphs have Boolean variables and their complements as terminal nodes, Boolean sum and product operations as nonterminal nodes, and edges from each operation to its arguments. The root node of the POG is denoted $\mathbf{r}$. In the CPOG (for "Certified POG") framework, a proof consists of a sequence of equivalence-preserving clause addition and deletion steps.

The equivalence proof for standard knowledge compilation has two phases. The *forward implication* phase serves to prove $\phi \Rightarrow \psi$, i.e., that any assignment satisfying the input formula also satisfies the compiled formula. It starts with the clauses for ϕ, along with clauses providing a Tseitin encoding of the compiled formula ψ, denoted $\theta(X, Z)$, where Z is the set of auxiliary variables introduced by the encoding. The proof continues with a sequence of added clauses, each of which must satisfy the *reverse unit propagation* (RUP) property [25, 43] with respect to the existing clauses. This property guarantees that the added clause is logically implied by the earlier clauses. The forward implication phase terminates with the addition of a unit clause $[r]$, where r is the variable in Z denoting root node $\mathbf{r}$. Adding this clause proves that any assignment α that satisfies ϕ must cause ψ to evaluate to 1. This phase can have exponential complexity. For example, when the input formula is unsatisfiable, the forward implication proof must be able to certify unsatisfiability.

The reverse implication phase for standard knowledge compilation serves to prove $\psi \Rightarrow \phi$, completing the proof of equivalence. It does so by showing that each input clause satisfies the RUP property with respect to the combination of $\theta(X, Z)$ and unit clause $[r]$. That is, any assignment that falsifies the clause must cause ψ to evaluate to 0 (false). This clause can therefore be deleted without affecting the set of satisfying assignments. This portion of the proof is guaranteed to have polynomial complexity, since clausal entailment can be tested efficiently for d-DNNF formulas [9, 18].

For certifying projected knowledge compilation, we can build on the CPOG framework. The forward implication phase requires no change to the framework and only small changes to how the proofs are generated. Having the phase culminate with the addition of unit clause $[r]$ proves that any pair of assignments α to the data variables and β to the quantified variables that satisfies input formula ϕ will cause compiled formula ψ to evaluate to 1. This result holds even when ψ contains only data variables and therefore depends only on assignment α.

On the other hand, it is less clear how to generalize the reverse implication phase to handle projection. The input clauses contain both data and quantified variables, and so the simple strategy of showing that each individual input clause is implied by the compiled formula (containing only data variables) will not suffice. Instead, we handle the existential quantification via a form of Skolemization [3], associating some assignment β to the quantified variables for each satisfying assignment α to the data variables. We do so by extending the POG representation to include *Skolem nodes*, representing partial assignments to the quantified variables. This "Skolem POG" then represents two d-DNNF formulas:

- Interpreting the Skolem nodes as tautologies gives the compiled representation $\psi(D)$. This version is used in the forward implication phase.
- By interpreting the Skolem nodes as Boolean products, the formula assigns values to both data and quantified variables. This makes it possible to deduce a Skolem assignment β for any satisfying data assignment α. This version serves as the basis for the input clause

deletion steps in the reverse implication phase.

The Skolem POG can be generated by a combination of minor modifications to the projected knowledge compiler and postprocessing to partition the Boolean products into product and Skolem nodes.

In a prototype implementation of a proof generator and checker for projected model compilation, we found that storing the clausal representation of a Skolem POG, plus the proof steps for a RUP derivation of the input clauses in the reverse implication phase, led to excessive file sizes for the proofs—the degree for many of the Skolem nodes can be in the thousands and the size of each RUP step is linear in the size of the Skolem POG. To reduce file sizes and to speed up proof generation and checking, we shifted responsibility for proving that the input clauses can be deleted from the proof generator to the proof checker. As a result of this shift, we can also have the checker make inferences directly on the Skolem POG, rather than on its clausal representation. Since the test for each input clause has linear complexity (in the size of the Skolem POG), this shift in responsibility still fulfills a requirement for propositional proof systems that the proof must be checkable in polynomial time, with respect to the size of the proof [14].

3 Overview and Prior Work

This paper describes how our earlier work on certifying the results of a standard knowledge compiler can be extended to those of a projected knowledge compiler. It provides background on the relevant logical foundations, the extension of the POG representation to include Skolem nodes, the structure of both the forward and the reverse implication phases, and a sketch of a soundness proof for the framework. We also discuss the use of HOL4 and CakeML to formally verify the soundness proof and to implement a formally verified proof checker.

We present experimental results for our implementation, consisting of modified versions of D4 and our earlier proof generation and checking tools. By shifting the reverse implication phase of the proof to the proof checker, our framework scales to formulas having large d-DNNF representations.

In terms of prior research, there has been some work on proof frameworks for model counting [5, 12, 23], along with comparisons of the expressive power of the different frameworks [4]. In addition, another proof framework has been developed for standard knowledge compilation [10]. We know of no previous work extending any of these frameworks to handle projection. Indeed, the literature on projected model counting and projected knowledge compilation is very sparse. Several algorithms and tools for projected knowledge compilation and model counting have been published, based on top-down approaches [1, 27, 31, 32, 35, 39], decision diagrams [20], and dynamic programming [22], but none of these can certify their results.

We have found that formal proof assistants are essential in uncovering the many subtle details that can arise in devising a sound proof framework. Working with a proof assistant also let us explore possible optimizations in the proof framework and checker with complete confidence of their soundness. In this work, we shifted from Lean 4 [19] to the HOL4 proof assistant [40]. We did so in part to take advantage of the CakeML programming language [29], which has a formally verified compiler [42] and a large body of support for developing verified machine-code implementations of propositional proof frameworks [24, 41].

Clause	File line	Formula	Comment
	`p cnf 4 3`		CNF formula with 4 variables and 3 clauses
	`c t pmc`		Projected model counting
	`c p show 1 2 0`	$D = \{a, b\}$	Declare data variables
1	`1 3 4 0`	$\overline{a} \Rightarrow (x \vee y)$	$\overline{a}$ triggers either x or y
2	`2 -3 0`	$x \Rightarrow b$	When x is triggered, force b to 1
3	`2 -4 0`	$y \Rightarrow b$	When y is triggered, force b to 1

Figure 1 CNF representation of disjunction of data variables a and b (numbered 1 and 2), with quantified variables x and y (numbered 3 and 4). Clause identifiers are implicitly assigned in the format and are shown here in red.

4 Logical Foundations

A Boolean formula ϕ over one or more disjoint sets of variables $X_1, X_2, \ldots, X_m$, is written as $\phi(X_1, X_2, \ldots, X_m)$ for clarity or as simply ϕ when the context is clear.

The *dependency set* for Boolean formula ϕ, denoted $\mathcal{V}(\phi)$, consists of all variables occurring in it. An *assignment* for a set of variables X is a mapping $\alpha\colon X \to \{0, 1\}$. When an assignment includes a value for every variable in $\mathcal{V}(\phi)$, it is said to be *total*; otherwise it is partial. We assume assignments are total unless explicitly stated. The set of all total assignments for X is denoted $\mathcal{U}(X)$. For formula ϕ with dependency set X, its set of *models*, denoted $\mathcal{M}(\phi)$ consists of all assignments in $\mathcal{U}(X)$ that satisfy the formula. For assignments $\alpha \in \mathcal{U}(X)$ and $\beta \in \mathcal{U}(Y)$, where $X \cap Y = \emptyset$, we write $\alpha{\cdot}\beta$ as the assignment in $\mathcal{U}(X \cup Y)$ assigning x to $\alpha(x)$ when $x \in X$ and to $\beta(x)$ when $x \in Y$. A *literal* is either a variable x or its complement $\overline{x}$. The symbol ℓ denotes a generic literal. Clauses are written as disjunctions, except that a unit clause, consisting of a single literal ℓ is written as $[\ell]$.

With a *projected formula* $\phi(D, Y)$ the variables are partitioned into a set of *data variables* D and a set of *quantified variables* Y. Its set of *data models*, denoted $\mathcal{M}_D(\phi)$ is defined as

$$\mathcal{M}_D(\phi) \;=\; \{\alpha \in \mathcal{U}(D) \mid \exists \beta \in \mathcal{U}(Y)\,.\,\alpha{\cdot}\beta \in \mathcal{M}(\phi)\} \tag{1}$$

The task of *projected knowledge compilation* is to convert a CNF formula $\phi(D, Y)$ into a formula $\psi(D)$ such that $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$. Furthermore, ψ should be in a form for which computing important properties, such as the number of models, can be performed in polynomial time, relative to the size of ψ.

The class of Boolean formulas known as *deterministic, decomposable, negation-normal form* (d-DNNF) is defined recursively as formulas ψ of the form:

Literals: x and $\overline{x}$ for Boolean variable x.

Decomposable Products [15, 18]:

$\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m$ where $\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_j) = \emptyset$ for $1 \leq i < j \leq m$

Deterministic Sums [16, 18]:

$\psi_1 \vee \psi_2 \vee \cdots \vee \psi_m$ where $\mathcal{M}(\psi_i) \cap \mathcal{M}(\psi_j) = \emptyset$ for $1 \leq i < j \leq m$. Here we assume for all $1 \leq i \leq m$, that $\mathcal{M}(\psi_i) \subseteq \mathcal{U}(X)$ for a common set of variables X.

We represent d-DNNF formulas as *partitioned-operation graphs* (POGs) [6, 7]. These are directed acyclic graphs where the terminal nodes represent literals and the nonterminal nodes represent product and sum operations. The root node is denoted $\mathbf{r}$.

As a running example throughout this paper, Figure 1 shows the declaration for a CNF formula encoding the disjunction of data variables a and b, with quantified variables x and

y. Following the DIMACS CNF notation, literals are represented as integers. In this case, we number the variables a and b as 1 and 2 and the variables x and y as 3 and 4. The two comment lines (beginning with 'c') indicate that the file encodes a projected model counting problem with variables 1 (a) and 2 (b) as the data variables. With clause 1, when a is assigned 0, either x or y must be assigned 1. With clauses 3 and 4, assigning 1 to either x or y will force b to be assigned 1.

5 Proof Requirements

Given a CNF formula $\phi(D, Y)$ and a d-DNNF formula $\psi(D)$, represented as a POG, we would like to generate a proof that $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$. We do so by converting ψ into clausal form as a CNF formula $\theta(D, Z)$, where Z is a set of *extension* variables used to encode the product and sum operations, according to a Tseitin encoding [38]. The *root variable* for the POG is defined as the extension variable r associated with root node $\mathbf{r}$. This encoding has the property that for any $\alpha \in \mathcal{U}(D)$, there is a unique assignment $\gamma \in \mathcal{U}(Z)$ such $\alpha{\cdot}\gamma \in \mathcal{M}(\theta)$. We refer to this assignment as γ_α. Furthermore, this assignment will satisfy $\gamma_\alpha(r) = 1$ if and only if $\alpha \in \mathcal{M}(\psi)$. We use the notation $\theta_r(D, Z)$ to represent the formula $\theta(D, Z) \cup \{[r]\}$, the combination of the Tseitin encoding of the POG and a unit clause with the root literal requiring the POG to evaluate to 1. Formula θ_r therefore serves as the clausal representation of d-DNNF formula ψ.

The proof of $\mathcal{M}(\psi) = \mathcal{M}_D(\phi)$ consists of two phases. The *forward implication* phase serves to prove that any data model α for $\phi(D, Y)$ will be a model for $\psi(D)$. With $\theta_r(D, Z)$ serving as the clausal representation of ψ, we can write this as

$$\forall\alpha\big[\exists\beta\,.\,\alpha{\cdot}\beta \in \mathcal{M}(\phi) \;\Rightarrow\; \exists\gamma\,.\,\alpha{\cdot}\gamma \in \mathcal{M}(\theta_r)\big] \tag{2}$$

We can transform this formula by bringing β outward and inverting the sense of the quantification. Furthermore, we can make use of the fact that γ is uniquely determined by α. These changes yield the formula

$$\forall\alpha\;\forall\beta\big[\alpha{\cdot}\beta \in \mathcal{M}(\phi) \;\Rightarrow\; \alpha{\cdot}\gamma_\alpha \in \mathcal{M}(\theta_r)\big] \tag{3}$$

We can see here that the assignments to both the data variables and the quantified variables are quantified in the same manner. We can therefore use the same method to generate and check this proof as is done with standard knowledge compilation [6, 7]. Starting with the combined clauses $\phi(D, Y)$ and $\theta(D, Z)$, the proof of (3) consists of a sequence of clause addition steps leading to the addition of the unit clause $[r]$. In doing so the proof transforms θ, the clausal representation of the POG operations, into θ_r, the clausal representation of the d-DNNF formula ψ.

The *reverse implication* phase serves to prove that, for any assignment α satisfying $\psi(D)$, there is a $\beta \in \mathcal{U}(Y)$ such that $\alpha{\cdot}\beta$ satisfies $\phi(D, Y)$. Working directly with the d-DNNF formula ψ, we can write this as:

$$\forall\alpha\big[\alpha \in \mathcal{M}(\psi) \;\Rightarrow\; \exists\beta\,.\,\alpha{\cdot}\beta \in \mathcal{M}(\phi)\big] \tag{4}$$

We can bring β outward, yielding

$$\forall\alpha\;\exists\beta\big[\alpha \in \mathcal{M}(\psi) \;\Rightarrow\; \alpha{\cdot}\beta \in \mathcal{M}(\phi)\big] \tag{5}$$

Here, the quantifications of α and β differ: they require the implication to hold for *all* data assignments α but for only *some* quantified assignment β for a given α—and even then only when α is a data model of ψ. The task of generating such an assignment is known as *Skolemization*, and β is referred to as a *Skolem assignment* for α [2, 3].

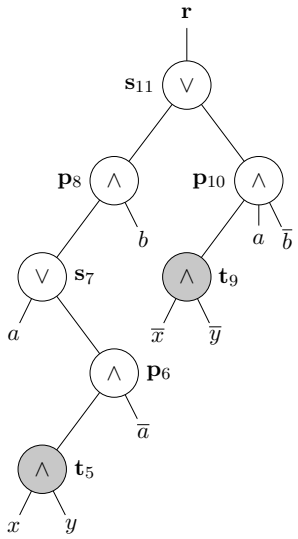

Figure 2 Skolem POG representing disjunction of data variables a and b, with quantified variables x and y. The Skolem nodes (shown in gray) define an assignment to the quantified variables for a given assignment to the data variables.

6 Skolem POGs

A Skolem POG is an extension of the POG representation that serves two roles:

Compiler Output: It provides the POG representation of a d-DNNF formula $\psi(D)$ forming the output of a projected knowledge compiler.

Skolemization: It augments this representation with a method for computing a Skolem assignment β for any assignment $\alpha \in \mathcal{M}(\psi)$ such that $\alpha \cdot \beta \in \mathcal{M}(\phi)$.

6.1 Skolem POG Example

Figure 2 illustrates a Skolem POG generated from the CNF formula $\phi(D, Y)$ of Figure 1 with $D = \{a, b\}$ and $Y = \{x, y\}$. As can be seen, the graph is itself a POG—every product operation is defined over disjoint sets of variables, and every sum operation is defined over disjoint sets of models. The gray nodes, known as *Skolem nodes*, indicate a special class of product operation. Skolem nodes only have literals of quantified variables as arguments, and they are the only nodes having such arguments.

The nodes shown in white comprise the d-DNNF generated by D4. We can see that D4 split on variable b at the top level, with node $\mathbf{p}_8$ representing the case where $\alpha(b) = 1$ and node $\mathbf{p}_{10}$ indicating the case where $\alpha(b) = 0$, with these being combined by the root node $\mathbf{s}_{11}$. On the left-hand side, we can see that $\mathbf{s}_7$ shows a split on variable a, resulting in the tautology $a \vee \overline{a}$. On the right-hand side, we see that node $\mathbf{p}_{10}$ requires $\alpha(a) = 1$ when $\alpha(b) = 0$ to satisfy the disjunction $a \vee b$.

The Skolem nodes provide Skolem assignments for some of the assignments to a and b. For $\alpha(b) = 1$ and $\alpha(a) = 0$, there are three possible assignments to variables x and y that satisfy the input clauses. Skolem node $\mathbf{t}_5$[3] indicates the choice $\beta(x) = \beta(y) = 1$. For $\alpha(b) = 0$

[3] The symbol "$\mathbf{t}$" is chosen to honor Thoralf Albert Skolem (1887–1963), the Norwegian logician for whom Skolemization is named.

and $\alpha(a) = 1$, there is a single satisfying assignment with $\beta(x) = \beta(y) = 0$, as indicated by Skolem node $\mathbf{t}_9$. We can see that no Skolem assignment is given for $\alpha(a) = \alpha(b) = 0$. This case is not needed, since it does not satisfy the input formula. We can also see that no Skolem assignment is given for $\alpha(a) = \alpha(b) = 1$. This is not required, since the data assignment alone guarantees that all input clauses are satisfied.

This example shows that, by considering or ignoring the Skolem nodes, a single graph can serve the dual roles of knowledge compiler output and Skolemization.

6.2 Clausal Encoding

In generating a clausal representation of a POG, we introduce an extension variable for each sum and product node. For a node with k children, we add $k + 1$ clauses, providing a Tseitin encoding of the operation over its arguments. Generating a clausal encoding of a Skolem POG is only required for the forward implication phase of the proof. We can therefore encode a Skolem node $\mathbf{t}$ as an extension variable t with a single unit clause $[t]$ indicating that the node should be interpreted as a tautology.

7 Proof Generation and Checking

Generating the Skolem POG and the forward implication proofs involves only modest changes to the knowledge compiler and to the proof generator. A more substantial change is required for the shift of the reverse implication phase to the proof checker.

7.1 Compiler Modification

We make use of the knowledge compiler D4, which already supports projected knowledge compilation. As with other top-down knowledge compilers [17, 36], D4 proceeds in a manner similar to a CDCL SAT solver. On each step, it performs unit propagations and simplifies the set of clauses by eliminating satisfied clauses and by removing falsified literals from the remaining clauses. When it selects an unassigned variable x, it must, in general, consider both assignments x and $\overline{x}$. These *splitting variables* become the basis for deterministic sums, with one argument satisfied only by assignments α with $\alpha(x) = 1$ and the other only with $\alpha(x) = 0$. The assigned and propagated literals then become arguments for decomposable product operations.

When running in projected compilation mode, D4 splits only on data variables. Once the simplified set of clauses contains only quantified variables, it calls a SAT solver to determine whether or not the formula is satisfiable. If so, it emits a tautology for the compiled result, and if not it emits a conflict. In addition, when unit propagation detects a unique assignment for a variable, it only includes the corresponding literal in the output for a data variable.

Modifying D4 to generate the information required to construct a Skolem POG involves two small changes:

1. After the SAT solver determines that a set of clauses containing only quantified variables is satisfiable, request a satisfying assignment. These become the arguments for a Skolem node. For example, node $\mathbf{t}_5$ in Figure 2 was generated after the assignment $\alpha(b) = 1$ and $\alpha(a) = 0$ reduced the input clauses in Figure 1 to the single clause $x \vee y$. The SAT solver detected that this clause was satisfiable and returned the assignment $\beta(x) = \beta(y) = 1$. This case only occurs as D4 reaches a terminal condition; it causes a Skolem node to be generated rather than a tautology.

2. Issue literals for both data and quantified variables when unit propagation occurs. These are then grouped into arguments for product and Skolem nodes. For example, assigning $\alpha(b) = 0$ to the clauses of Figure 1 first causes the unit propagations $\beta(x) = \beta(y) = 0$ and then $\alpha(a) = 1$. These assignments are grouped into nodes $\mathbf{t}_9$ and $\mathbf{p}_{10}$ in Figure 2. This case can occur at any point where unit propagation occurs; it can cause a Skolem node to be generated as the argument to a product node at upper levels in the POG.

Our modified version of D4 issues the additional literals as part of its regular output. We postprocess these into the required structure for a Skolem POG.

7.2 Proof Generation

The *monolithic* proof generation method devised for standard knowledge compilation [6] can be adapted for projected knowledge compilation. The generator runs a proof-generating SAT solver on the (hopefully) unsatisfiable CNF formula consisting of the clauses in formulas $\phi(D, Y)$ and $\theta(D, Z)$, as well as the unit clause $[\overline{r}]$. The solver effectively proves that the POG cannot generate value 0 as its output for any data assignment $\alpha \in \mathcal{M}_D(\phi)$. Each of the proof clauses is then augmented with the literal r, and so the empty clause of the proof becomes the unit clause $[r]$. This approach works regardless of any preprocessing transformations made by the knowledge compiler.

We also support a *hybrid* proof generation method, where the proof generator recursively traverses the structure of the POG, as with a structural proof generation method [7], but stopping once the size of a subgraph is sufficiently small. The proof for the subgraph is generated monolithically. This approach only works if the recursive traversal of the d-DNNF formula generated by the knowledge compiler yields corresponding simplifications and unit propagations of the CNF input formula. This may not hold if the CNF has been transformed by preprocessing.

7.3 Proof Checking

We have shifted the responsibility for the reverse implication phase to the proof checker. Letting $\hat{\psi}(D, Y)$ denote the d-DNNF formula encoded by the Skolem POG, the checker should ensure that $\hat{\psi}(D, Y) \Rightarrow \phi(D, Y)$. It can so by checking that for each input clause C, any assignment to the data and quantified variables that falsifies this clause will also cause $\hat{\psi}$ to evaluate to 0. It does this by performing a bottom-up unit propagation through the Skolem POG. Figure 3 demonstrates how this process works for the three input clauses from the formula of Figure 1. In each case, we can see that negating the literals of the clause causes some of the arguments to the Skolem POG operations to be assigned value 0. These are shown as dashed lines. Arguments that are assigned value 1, as well as unassigned arguments are shown as solid lines. A product or Skolem operation having an argument assigned value 0 will also evaluate to 0, and a sum operation will evaluate to 0 when both arguments evaluate to 0. The result in all three cases is to assign 0 to root variable r.

Section 8 presents a soundness proof for the forward and reverse implication phases.

7.4 SCPOG Example

Our SCPOG proof framework modifies and extends the earlier CPOG framework to enable certifying the results of a projected model counter. A single SCPOG file describes both a Skolem POG representation of a formula, as well as the forward implication proof. Here, we present the main features of the file format via an example. A more comprehensive presentation is included as Appendix A.

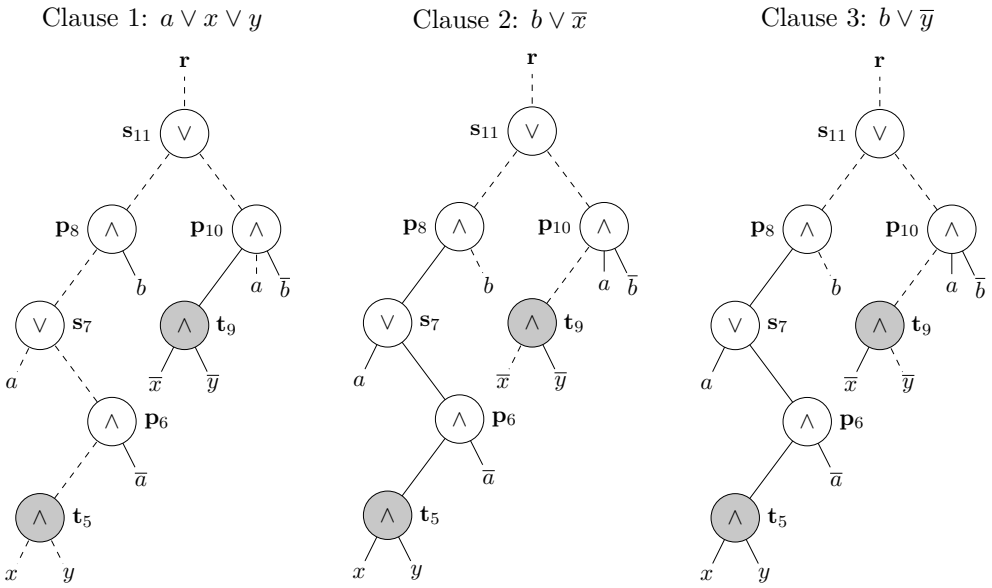

Figure 3 Justifying input clause deletions. Negating the clause literals and then unit propagating through the operations cause the Skolem POG to evaluate to 0. Literals assigned value 0 are shown as dashed lines.

Figure 4(A) shows the complete SCPOG file for the example with the CNF file shown in Figure 1 and the Skolem POG shown in Figure 2. The CNF has four variables and three clauses, and so it starts numbering extension variables with 5 and added clauses with ID 4. The first line declares the root variable r as the extension variable we denote as s_{11}. The next seven lines declare the Skolem POG operations, as shown in Figure 2. These implicitly generate the set of clauses shown in Figure 4(B). Skolem nodes $\mathbf{t}_5$ and $\mathbf{t}_9$ require just single (unit) clauses, while the other node declarations implicitly add either three or four clauses, and so the clause identifiers in the declarations must be spaced apart.

Every sum node must include a mutual exclusion proof to certify determinism. Each proof is given as list of clauses providing a RUP proof that the two arguments cannot both evaluate to 1. In the example, the declaration of sum node $\mathbf{s}_7$ lists clause 6 ($\overline{p}_6 \vee \overline{a}$) as proof that arguments p_6 and a cannot both evaluate to 1. The declaration of sum node $\mathbf{s}_{11}$ references clauses 12 ($\overline{p}_8 \vee b$) and 17 ($\overline{p}_{10} \vee \overline{b}$) for the mutual exclusion proof. These resolve to the clause $\overline{p}_8 \vee \overline{p}_{10}$, proving that arguments p_8 and p_{10} cannot both evaluate to 1.

Clause IDs 22–28 show the forward implication proof leading to the addition of unit clause $[s_{11}]$ as the final step.

Overall, we can see that this file contains a full documentation of the d-DNNF formula generated by the knowledge compiler, the Skolem assignments to the quantified variables associated with the assignments to the data variables, and a forward implication proof.

In addition to checking the mutual exclusion and forward-implication proofs, the proof checker must ensure that the arguments to each product operation are decomposable, and it must check that reverse implication holds.

(A) SCPOG Declaration

ID	File line	Formula
Root declaration		
	`r 11`	$r = s_{11}$
Skolem POG declaration		
4	`t 5 3 4 0`	$t_5 = x \wedge y$
5	`p 6 -1 5 0`	$p_6 = \overline{a} \wedge t_5$
8	`s 7 6 1 6 0`	$s_7 = p_6 \wedge a$
11	`p 8 2 7 0`	$p_8 = b \wedge s_7$
14	`t 9 -3 -4 0`	$t_9 = \overline{x} \wedge \overline{y}$
15	`p 10 1 -2 9 0`	$p_{10} = a \wedge \overline{b} \wedge t_9$
19	`s 11 8 10 12 17 0`	$s_{11} = p_8 \vee p_{10}$
Forward implication proof		
22	`6 -2 1 0 4 5 0`	$b \wedge \overline{a} \Rightarrow p_6$
23	`7 -2 0 9 22 10 0`	$b \Rightarrow s_7$
24	`8 -2 0 23 11 0`	$b \Rightarrow p_8$
25	`1 2 0 3 2 1 0`	$a \vee b$
26	`10 2 0 25 14 15 0`	$\overline{b} \Rightarrow p_{10}$
27	`-2 11 0 20 24 0`	$b \Rightarrow s_{11}$
28	`11 0 27 21 26 0`	$[s_{11}]$

(B) Implicit Definitions

ID	Clauses	Node
4	`5 0`	$\mathbf{t}_5$
5	`6 1 -5 0`	$\mathbf{p}_6$
6	`-6 -1 0`	
7	`-6 5 0`	
8	`-7 6 1 0`	$\mathbf{s}_7$
9	`7 -6 0`	
10	`7 -1 0`	
11	`8 -2 -7 0`	$\mathbf{p}_8$
12	`-8 2 0`	
13	`-8 7 0`	
14	`9 0`	$\mathbf{t}_9$
15	`10 -1 2 -9 0`	$\mathbf{p}_{10}$
16	`-10 1 0`	
17	`-10 -2 0`	
18	`-10 9 0`	
19	`-11 8 10 0`	$\mathbf{s}_{11}$
20	`11 -8 0`	
21	`11 -10 0`	

Figure 4 SCPOG Example. A single file (A) declares the Skolem POG structure and provides the forward implication proof. The declaration of the Skolem POG implicitly defines a set of clauses (B). Clause IDs are shown in red.

8 Soundness

We have formally verified the soundness of our proof framework in HOL4 and developed a verified proof checker using CakeML. Here, we provide an informal justification of the proof framework to give some intuition for its soundness. Section 8.2 provides more details on the formally verified checker.

8.1 Informal Justification

As a recap of notation, the POG has two forms: $\psi(D)$, including the sum and product nodes, with the Skolem nodes treated as tautologies, and $\hat{\psi}(D, Y)$, where the Skolem nodes are treated as Boolean products of their arguments (which are literals of quantified variables). The operations in POG formula $\psi(D)$ have Tseitin clausal representation $\theta(D, Z)$, and when we add unit clause $[r]$, we obtain $\theta_r(D, Z)$, the clausal representation of $\psi(D)$.

As further notation, we require a way to describe an implication relation between formulas over different sets of variables, considering only some of the variables they have in common. For formulas $\phi_1(S, T_1)$ and $\phi_2(S, T_2)$, define the relation $\phi_1 \Rightarrow_S \phi_2$ as holding when $\mathcal{M}_S(\phi_1) \subseteq \mathcal{M}_S(\phi_2)$. This notation makes no assumption about whether or not sets T_1 and T_2 are disjoint. We can see that for formulas $\phi_1(X)$ and $\phi_2(X)$, the standard implication $\phi_1 \Rightarrow \phi_2$ is equivalent to $\phi_1 \Rightarrow_X \phi_2$.

The following propositions state two key properties of this relation: narrowing the

set of variables preserves implication, and dropping conjuncts in the consequent preserves implication. Note that in conjunctive normal form, the conjunction of two formulas can be expressed as the union of their clauses.

▶ **Proposition 1.** *For formulas $\phi_1(S, T_1)$, $\phi_2(S, T_2)$, if $\phi_1 \Rightarrow_S \phi_2$, then any subset $S' \subseteq S$ maintains the implication $\phi_1 \Rightarrow_{S'} \phi_2$.*

▶ **Proposition 2.** *For formulas $\phi_1(S, T_1)$, $\phi_2(S, T_2)$, and $\phi_3(S, T_3)$, if $\phi_1 \Rightarrow_S \phi_2 \wedge \phi_3$, then $\phi_1 \Rightarrow_S \phi_2$.*

The forward implication proof must show that $\phi(D, Y) \Rightarrow_D \theta_r(D, Z)$, fulfilling the requirement for the forward implication property (2). It begins with the clauses $\theta^1(D, Y, Z) \doteq \phi(D, Y) \cup \theta(D, Z)$, the combination of the input formula and the Tseitin encoding of the POG operations.[4] Since the union of two sets of clauses is equivalent to their conjunction, $\theta^1 = \phi \wedge \theta$. Furthermore, $\theta(D, Z)$ induces an assignment $\gamma_\alpha \in \mathcal{U}(Z)$ for every $\alpha \in \mathcal{U}(D)$, such that $\alpha \cdot \gamma_\alpha$ satisfies θ. Therefore appending θ to ϕ does not reduce the set of data models. That is, $\phi(D, Y) \Rightarrow_{D \cup Y} \theta^1(D, Y, Z)$, and therefore by Proposition 1, $\phi(D, Y) \Rightarrow_D \theta^1(D, Y, Z)$.

The RUP addition steps define a sequence of formulas $\theta^1, \theta^2, \ldots, \theta^s$ concluding with $\theta^s(D, Y, Z) \supseteq \phi(D, Y) \cup \theta(D, Z) \cup \{[r]\} \equiv \phi(D, Y) \wedge \theta_r(D, Z)$. Each proof step preserves implication: $\theta^i \Rightarrow_D \theta^{i+1}$ for $1 \leq i < s$, and therefore $\theta^1 \Rightarrow_D \theta^s$. By Proposition 2, we can see that these steps provide a proof that $\phi(D, Y) \Rightarrow_D \theta_r(D, Z)$, and therefore $\phi(D, Y) \Rightarrow_D \psi(D)$.

The reverse implication phase involves making sure that $\hat{\psi}(D, Y) \Rightarrow_D \phi(D, Y)$, fulfilling property (4). The proof checker does so by showing that $\neg C \Rightarrow \neg\hat{\psi}$, and therefore $\hat{\psi} \Rightarrow C$ for each clause $C \in \phi$. Combining all of these checks proves that $\hat{\psi}(D, Y) \Rightarrow \phi(D, Y)$, and therefore $\hat{\psi}(D, Y) \Rightarrow_D \phi(D, Y)$.

The combination of the forward and reverse implications shows that the two formulas for the Skolem POG bracket the input formula:

$$\hat{\psi}(D, Y) \;\Rightarrow_D\; \phi(D, Y) \;\Rightarrow_D\; \psi(D)$$

In terms of the data models, we can see therefore that $\hat{\psi}$ provides a lower bound, while ψ provides an upper bound: $\mathcal{M}_D(\hat{\psi}) \subseteq \mathcal{M}_D(\phi) \subseteq \mathcal{M}_D(\psi)$.

Importantly, we can also see that the two formulas for the Skolem POG are equivalent in terms of their data models:

▶ **Proposition 3.** *A Skolem POG with representations $\hat{\psi}(D, Y)$ and $\psi(D)$ has $\mathcal{M}_D(\hat{\psi}) = \mathcal{M}_D(\psi)$.*

This equivalence is proved by induction over the POG structure. The key observation is that the decomposability of each product operation makes it possible to derive the Skolem assignments for its arguments independently. Our bracketing is therefore an equivalence: $\mathcal{M}_D(\hat{\psi}) = \mathcal{M}_D(\phi) = \mathcal{M}_D(\psi)$, completing the proof that the POG formula $\psi(D)$ serves as the projected compilation of input formula $\phi(D, Y)$.

8.2 End-to-End Verification

We have formally verified the proof framework and used a refinement-based process to develop a formally verified CakeML implementation of the SCPOG proof checker in HOL4. This

[4] In our proof format specification (Appendix A), SCPOG proof steps are allowed to arbitrarily interleave RUP additions and node declaration steps; the soundness argument is similar.

is broadly similar to the development of verified machine-code implementations of prior propositional proof frameworks [24, 41].

Checking proofs in these other frameworks only requires checking a sequence of clause additions. The SCPOG proof format also requires checking the decomposability and determinism of the specified POG and to check each input clause for the reverse implication. The CakeML checker starts with a clausal representation of the SCPOG nodes when checking the forward implication proof. Upon success of this phase, it internally converts this representation into an equivalent adjacency matrix-like representation where all nodes and their children can be looked up in constant time. For each input clause C, this allows the CakeML checker to run a fast, top-down evaluation from root node $\mathbf{r}$ to check that the POG evaluates to 0 under the assignment corresponding to $\neg C$; the checker short-circuits evaluation wherever possible (e.g., when a child of a product node is falsified, it does not recurse into other children). The constant-time lookup representation is particularly important for Skolem nodes, which can contain a large number of quantified literals. For such nodes, we iterate over the literals in C and check if any of them are in the Skolem node.

The final correctness theorem for the CakeML proof checker is obtained by compiling its verified source implementation with CakeML's formally verified compiler [42]. Informally, the resulting verified theorem states that:

▶ **Proposition 4.** *Assuming the SCPOG checker's machine code is executed on a system satisfying the standard CakeML assumptions (for the x64 ISA), then:*

- *if it prints out* `"s VERIFIED UNSAT"` *on standard output, then the input CNF file parsed to an unsatisfiable CNF formula;*
- *if it prints out* `"s VERIFIED CPOG REPRESENTATION"` *on standard output, then the input CNF file parsed to a CNF formula and the input proof file contains a POG such that the CNF and POG have the same set of models on the specified data variables; moreover, the CPOG indeed represents a d-DNNF, i.e., it is decomposable and deterministic; and*
- *all proof checking failures lead to error messages printed to standard error.*

9 Experimental Results

We implemented a proof generator in C++, along with two proof checkers: one written in C, and the other in CakeML and verified with HOL4. The C checker uses multithreading when performing the reverse-implication check, taking advantage of the property that the input clause deletion checks can be performed independently.

All experiments were run on a 2021 Apple MacBook Pro, with a 3.2 Ghz Apple M1 8-core processor and 64 GB of RAM. All reported times are for elapsed ("wall-clock") times.

For benchmark problems, we used the private and public formulas from the 2024 unweighted and weighted projected model counting competitions.[5] There were 200 formulas for each track, but with some duplicates across the two tracks, yielding a set of 389 unique benchmark formulas.

We ran two versions of D4 on the 389 benchmark formulas: an unmodified version generating just the d-DNNF data, and the modified version that included the information from which the Skolem POG could be constructed. Both successfully compiled 175 formulas within the time limit of 1,000 seconds. The modified version of D4 ran fast enough: at most 1.8× slower than the unmodified version, with a median of 1.11×.

[5] Available at `https://zenodo.org/records/14249068`.

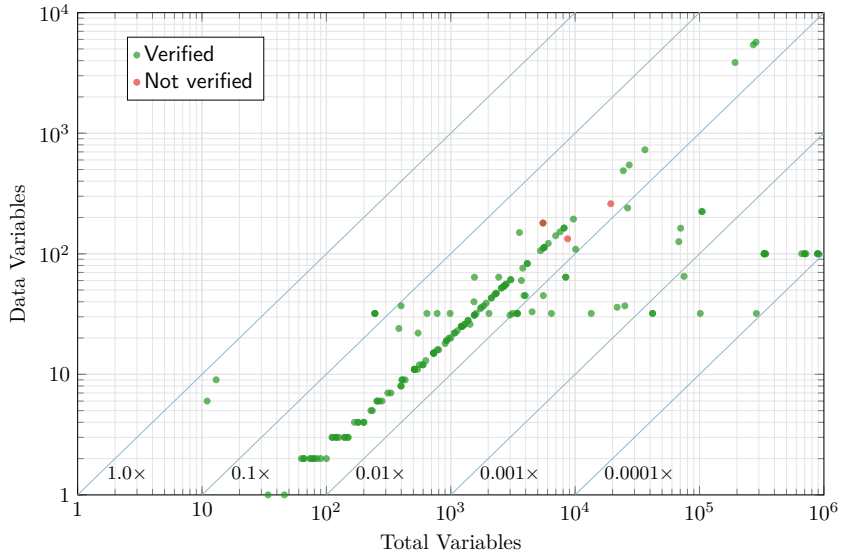

Figure 5 Data Variables vs. Total Variables. The median fraction was 2.0%. Colors indicate the level of success achieved in verification.

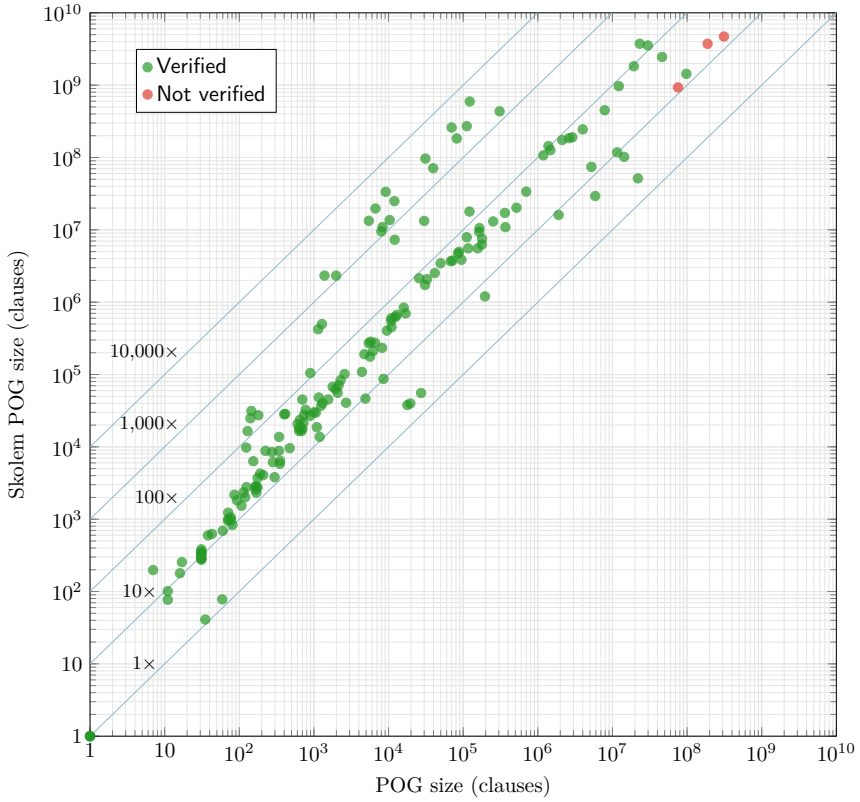

Figure 6 Skolem POG overhead. Skolem POG size vs. POG size. The median size ratio was 34×. Colors indicate the level of success achieved in verification.

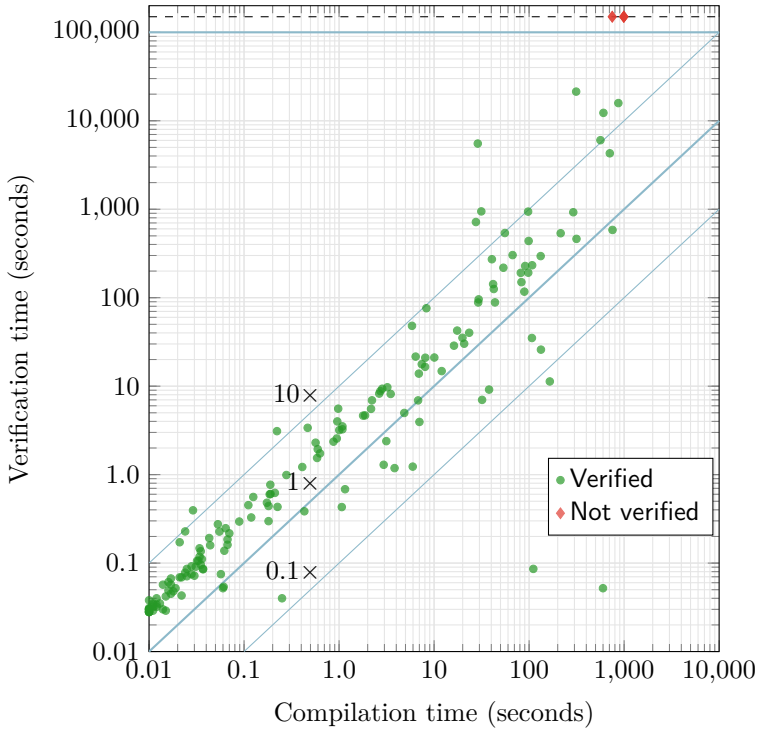

Figure 7 Verification vs. compilation time. Verification includes proof generation and checking. The reverse implication check uses 7-way multithreading.

Figure 5 documents an important feature of the 175 compiled benchmark formulas, showing the number of data variables (Y-axis), compared to the total number of variables (X-axis). For many formulas, the data variables constitute only a small fraction of the total. For one, only 100 of its 906,806 (0.0001%) variables are data variables. The median ratio of data variables to total variables is 2.0%. This property has significant impact on the sizes of the Skolem POGs and the challenge of checking reverse implication. The color coding of the data in the plots of Figure 5 and 6 gives a preview of our success in verifying these benchmarks: 172 of them (green) completed the full verification. For the remaining 3 (red), the proof generator timed out, with a time limit of 10,000 seconds.

We generated the Skolem POGs from the output of the modified version of D4, and we could also determine how large the POG would be without the Skolem information, as illustrated on the Y and X axes of Figure 6, respectively. We measure POG sizes in terms of the number of clauses in their Tseitin encodings. This is also equal to the number of nodes plus the number of edges in the graphs. As can be seen, the Skolem POGs tend to me much larger than the regular POGs, with size ratios ranging up to 4857× with a median of 34×. Considering the large fraction of quantified variables in these formulas, these large size ratios are to be expected.

Figure 7 compares the runtime for verification versus for compilation for the 172 formulas that were fully verified. The X axis shows the time to run the modified version of D4, while the Y axis shows the time to postprocess the D4 output, generate and check the proof, and compute an unweighted or weighted model count. The reverse-implication check uses 7-way multithreading. For 19 formulas, compilation took longer than verification, while it required less for the others. At worst, verification required 191.7× longer, with a median of 2.95×.

Our use of multithreading provides a significant benefit. Verifying all 172 benchmarks requires a total of 20.8 hours with the multithreaded version and 37.9 hours with a sequential version. This includes proof generation, forward and reverse checking, and model counting. Parallelism is only used when performing the reverse-implication check on the input clauses. This portion requires 8.4 hours with multithreading and 25.5 hours sequentially, a speedup of 3.0×. This is a reasonable performance gain for a memory-intensive application. Multithreading also reduces the benchmark with the longest sequential verification time from 15.3 to 4.4 hours.

We ran the formally verified checker on the 172 formulas for which the proof generator completed. It verified 156 of them, and it exceeded its heap memory limit of 48 GB for the remaining 16. Comparing its performance to the unverified checker for the 156 completed checks, it ran between 2.9× faster and 17.7× slower, with a median of 1.5× slower. This is remarkable performance for a formally verified tool.

Overall, our tool handled these formulas well. On the other hand, we found that letting D4 run for longer than 1000 seconds leads to d-DNNF formulas that are beyond the capacity of our current tool chain. The major weakness is in generating forward implication proofs.

10 Conclusions and Further Research

We have shown that the output of a projected knowledge compiler can be certified by extending our prior proof framework for standard knowledge compilation. The compiler already has access to information about how the quantified variables can be assigned according to the assignment to the data variables. By having it provide this information and then postprocessing the output, the proof generator can construct a Skolem POG, serving the dual roles of encoding the projected formula and the Skolem assignments. Certification proceeds by proving implication in both directions, within forward implication proved by clausal additions and reverse implication checked directly on the generated formula.

We were able to formally verify the soundness of our proof framework in HOL4 and develop a formally verified checker. A successful run of the checker gives full confidence that the compiled formula $\psi(D)$ has the same data models as the input formula $\theta(D, Y)$. In our earlier effort to devise a proof framework for standard knowledge compilation, the formal verification effort uncovered some subtle requirements that we had overlooked. Our experience was quite different this time. We correctly anticipated the requirements to achieve soundness. On the other hand, having the support of formal verification enabled us to be more aggressive in exploring refinements to improve the combined proof generator and checker performance. These optimizations greatly improved the scaling of our tools.

Projected knowledge compilation, and the closely related topic of projected model counting, are active areas of research, with many optimizations being discovered. Our framework will only be of value if it can keep up with these innovations. We describe two examples here.

First, some have discovered conditions under which it is possible for the compiler or model counter to split on quantified variables, remove the literals for these variable in the compiled output, and still have the result be a d-DNNF formula [27]. This requires more extensive mutual exclusion proofs for the sum operations than the simple cases illustrated in the SCPOG example of Figure 4. Our framework, described in Appendix A, includes support for such proofs. For a sum node with children represented by literals ℓ_1 and ℓ_2, the mutual exclusion proof can be generated with a proof-generating SAT solver that includes clauses encoding the two branches, plus the unit literals $[\ell_1]$ and $[\ell_2]$. The proof clauses in the UNSAT proof are augmented by the literals $\overline{\ell}_1$ and $\overline{\ell}_2$, and so the empty clause of the

Table 1 SCPOG Clause Types. Different types have different rules for generating them.

Clause Addition			
Type	How Generated	Variables	Hints
Input	Input formula	Data, Quantified	None
Tseitin	Sum and Product operations	Data, Extension	None
Tseitin	Skolem operations	Extension	None
Structural	RUP addition	Data, Extension	Tseitin, Structural
Forward	RUP addition	Data, Extension	Input, Tseitin, Structural

proof becomes the mutual exclusion $\overline{\ell}_1 \vee \overline{\ell}_2$.

Second, some have discovered cases where input clauses can be removed via *blocked-clause elimination* [32]. Significantly, they do so as an inprocessing step, where the conditions for the elimination hold only under a partial assignment to the variables. Our framework can handle the special case of *pure-literal elimination*, where the blocked clause has been reduced to a unit literal, by having the knowledge compiler record the literal as being satisfied. Handling more general cases would seem to require a new way to prove reverse implication for those input clauses that have been eliminated.

A The SCPOG Format

This appendix documents the SCPOG proof format. A single file declares the structure of the Skolem POG and provides the required proof steps. The file consists of a sequence of steps that either implicitly or explicitly add clauses. The clauses in the input CNF file are not in the SCPOG file, but they serve as part of the proofs. Overall, the clauses support two different proof types: the *mutual exclusion* proofs for the sum operations, guaranteeing that these operations are deterministic, and the *forward implication* proof that the generated POG is implied by the quantified input formula.

The variables are partitioned into three disjoint classes:

Data: The variables declared as "show" variables in the input CNF file

Quantified: All other variables occurring in the input CNF file

Extension: The variables associated with the POG Sum, Product, and Skolem operations

The clauses are also partitioned into disjoint classes. This categorization is determined by how they are generated and how they are used.

Input: The clauses in the input CNF file

Tseitin: The clauses implicitly defined by the Sum, Product, and Skolem operations

Structural: Clauses to express the mutual exclusion proofs for the Sum operations

Forward: Clauses added as steps in the forward implication proof

Table 1 provides more detail about the different clause types, and how these clauses are generated. According to its type, each clause has restrictions on how it is generated, what variables it contains, and what clauses can serve in the hints to support their addition.

- *Input* clauses contain only data and quantified variables. Their addition is implicit.
- *Tseitin* clauses encode the sum, product, and Skolem operations, each defining an extension variable in terms of data and other extension variables. Each Skolem operation is encoded by a single unit clause.

Table 2 SCPOG Step Types. C: clause identifier, L: literal, V: variable

Rule				Description
	`r`	V		Declare root variable
	`r`	`0`		Declare unsatisfiable formula
C	`a`	L^* `0`	C^+ `0`	Add Forward clause
C	`as`	L^* `0`	C^+ `0`	Add Structural clause
C	`p`	V L^* `0`		Declare product operation
C	`s`	V L L	C^+ `0`	Declare sum operation
C	`t`	V L^* `0`		Declare Skolem operation

- *Structural* clauses support the mutual exclusion proofs for the sum operations. They are generated via RUP addition starting with Tseitin clauses. They contain only data and extension variables. These clauses are only needed when the POG contains sum operations that are not decision nodes.[6]
- Forward clauses form the steps in the forward implication proof. They are added as RUP steps, starting with Input, Tseitin, and Structural clauses. They can contain only data and extension variables. The root clause is the final clause in the forward-implication proof. It consists of the unit clause $[r]$ for root variable r.

As can be seen, there are some ambiguities in how a clause should be classified. For example, a clause containing only data and extension variables and having only Tseitin and Structural clauses as hints could be a Structural or a Forward clause. We therefore have a distinct command for adding Structural clauses.

A.1 Syntax

Table 2 shows the declarations that can occur in a SCPOG file. The checker is provided with the input formula as a separate file. As with other clausal proof formats, a variable is represented by a positive integer v, with the first ones being input variables, and higher ones serving as 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 `r` command declares the root of the POG. For a satisfiable formula, it declares the root as a variable $r > 0$, indicating an extension variable representing the root of a graph. It must be declared before the root clause is added. For an unsatisfiable formula, the root declaration has value 0, and there is no root clause.

The second set of proof rules allows different clause types to be added. Forward clauses are added via RUP addition (command `a`), with a sequence of antecedent clauses (the "hint"). Structural clauses are added similarly, but with command `as`.

[6] A sum operation is a decision node when, for some variable x, the operation has one argument that is either the literal x or a product operation having x as an argument, while the other child is either the literal $\overline{x}$, or it is a product operation containing having literal $\overline{x}$ as an argument.

Table 3 Defining Clauses for Product (A) and Sum (B) Operations, and Skolem (C) Operations

(A). Product Operation

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

(B). Sum Operation

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

(C). Skolem Operation

ID	Clause
i	v

The final three step types declare the product, sum, and Skolem operations. The clauses implicitly generated by these operations are shown in Table 3.

The declaration of a *product* operation has the form:

$$i \quad \texttt{p} \quad v \quad \ell_1 \quad \ell_2 \quad \cdots \quad \ell_m \quad \texttt{0}$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \ldots, \ell_m$ is a sequence of m integers, indicating the arguments as literals of existing variables. The variables associated with these literals must be data variables or previously defined extension variables. To guarantee negation-normal form, any argument that represents some other operation must be positive. This declaration implicitly causes $m + 1$ Tseitin clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the product of its arguments. These clauses are shown in Table 3(A).

To guarantee decomposability, the dependency sets for the arguments represented by each pair of literals ℓ_i and ℓ_j must be disjoint, for $1 \leq i < j \leq k$. The dependency set for the operation then consists of the union of the dependency sets for its arguments. A product operation may have no arguments, representing Boolean constant 1. The only clause added to the proof will be the unit literal v.

The declaration of a *sum* operation has the form:

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

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. The variables associated with these literals must be data variables or previously defined extension variables. To guarantee negation-normal form, any argument that represents some other operation must be positive. Hint H consists of a sequence of clause IDs, all of which must be structural clauses.[7] The dependency set for the operation consists of the union of the dependency sets for its arguments. As Table 3(B) shows, this declaration implicitly causes three Tseitin clauses to be added to the proof, providing a Tseitin encoding that defines extension variable v as the sum of its arguments. The hint must provide a RUP proof of the clause $\overline{\ell}_1 \vee \overline{\ell}_2$, showing that the two arguments have disjoint models.

The declaration of a *Skolem* operation has the same form as a product operation:

[7] The restriction to structural 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 earlier efforts at formal verification.

$$i \quad \texttt{t} \quad v \quad \ell_1 \quad \ell_2 \quad \cdots \quad \ell_m \quad \texttt{0}$$

Integer i is a new clause ID, v is a positive integer that does not correspond to any previous variable, and $\ell_1, \ell_2, \ldots, \ell_m$ is a sequence of m integers, indicating the arguments as literals of existing variables. The variables associated with these literals must be quantified variables. This declaration implicitly causes a single clauses to be added to the proof, as is documented in Table 3(C). It is a unit clause indicating that the operation evaluates to 1 during the forward implication proof. The dependency set for the operation is the set of variables associated with the argument literals.

A.2 Overall Proof Requirements

We classify a compiled formula as *special* when it is unsatisfiable or a tautology. Otherwise it is *standard*. We describe the proof requirements for the special cases after we cover the standard ones. For standard cases, an SCPOG proof consists of the following:

- A declaration of the root variable.
- Declarations of POG sum, product, and Skolem operations
- RUP clause additions

Each of these steps must obey its respective rules. For a standard formula, the proof must, via a sequence of clause additions, generate the root clause $[r]$.

The standard case includes some formulas that might seem to require special consideration. A CNF formula containing only the unit clause $[\ell]$, for literal ℓ, can be compiled into a POG consisting of a product node with ℓ as its only argument. The forward implication proof can then proceed in the standard form. With projected knowledge compilation, a formula can become that of a single literal via projection. For example, the formula $(a \vee x) \wedge (a \vee \overline{x})$, when projected for quantified variable x becomes a. In this case, however, the Skolem POG for the formula will contain several nodes and satisfy the conditions for the standard cases.

A.3 Special Cases

For standard knowledge compilation, a formula can be a tautology only if it either contains no clause, or if every input clause contains some variable a and its complement $\overline{a}$. The POG representation consists of a single product node with no arguments, encoding constant 1. The declaration of this node yields an extension variable v that serves as the root variable: $r = v$. The unit clause for this variable is implicitly added by the declaration of the product operation. There is no need for a forward implication proof, since the implication holds trivially.

With projected knowledge compilation, a formula can become a tautology via projection. For example, the formula $(a \vee x) \wedge (\overline{a} \vee \overline{x})$, when projected for quantified variable x becomes $(a \vee \overline{a})$. In this case, however, the Skolem POG of this formula will contain several nodes and satisfy the conditions for the standard cases.

As is shown in Table 2, the SCPOG format representation of an unsatisfiable formula has a root declaration with value 0, rather than a literal. The forward implication proof consists of a sequence of clause addition steps leading to the addition of the empty clause. This special case holds for both standard and projected knowledge compilation—an unsatisfiable formula cannot become satisfiable via projection, nor can a satisfiable formula become unsatisfiable via projection.

References

1 Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *LNCS*, pages 121–137, 2015.

2 Valeriy Balabanov and Jie-Hong Roland Jiang. Unified QBF certification and its applications. *Formal Methods in Systems Design*, 41:45–65, 2012.

3 Marco Benedetti. Evaluating QBFs via symbolic Skolemization. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 3452 of *LNCS*, pages 285–300, 2005.

4 Olaf Beyersdorff, Johannes K. Fichte, Markus Hecher, Tim Hoffmann, and Kasche Kaspar. The relative strength of #SAT proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2024.

5 Olaf Beyersdorff, Tim Hoffman, and Luc Nicolas Spachmann. Proof complexity of propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2023.

6 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, July 2023.

7 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to formally verified model counting. *Journal of Artificial Intelligence Research*, 82:2057–2099, 2025.

8 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

9 Florent Capelli. Knowledge compilation languages as proof systems. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *LNCS*, pages 90–91, 2019.

10 Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-DNNF compilers. In *AAAI Conference on Artificial Intelligence*, 2021.

11 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172:772–799, 2008.

12 Sravanthi Chede, Leroy N. Chew, and Anil Shukla. Circuits, proofs, and propositional model counting. In *Foundations of Software Technology and Theoretical Computer Science (FTTCS)*, 2024.

13 Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Symposium on Logic in Computer Science (LICS)*, pages 353–362. IEEE, 1989.

14 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 2014.

15 Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

16 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applications of Non Classical Logics*, 11(1-2):11–34, 2001.

17 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In *European Conference on Artificial Intelligence*, pages 328–332, 2004.

18 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17, 2002.

19 Leonardo de Moura and Sebastian Ulrich. The Lean 4 theorem prover and programming language. In *Conference on Automated Deduction (CADE)*, volume 12699 of *LNAI*, pages 625–635, 2021.

20 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. ProCount: Weighted projected model counting with graded project-join trees. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 12831 of *LNCS*, pages 152–170, 2021.

21 Thomas Eiter and Gabriele Kern-Isberner. A brief survey of forgetting from a knowledge representation and reasoning perspective. *Künstliche Intelligenz*, 33:9–33, 2019.

22 Johannes K. Fichte, Markus Hecher, Michael Morak, Patrick Thier, and Stefan Woltran. Solving projected model counting by utilizing treewidth and its limits. *Artificial Intelligence*, 314, 2023.

23 Johannes K Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl, 2022.

24 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *AAAI Conference on Artificial Intelligence*, pages 8038–8047, 2024.

25 Evgueni I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE)*, pages 886–891, 2003.

26 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. IOS Press, 2009.

27 Rafael Kiesel and Thomas Eiter. Knowledge compilation and more with SharpSAT-TD. In *Principles of Knowledge Representation and Reasoning*, pages 406–416, 2023.

28 Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, July 2017.

29 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

30 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 667–673, 2017.

31 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *AAAI Conference on Artificial Intelligence*, pages 1536–1543, 2019.

32 Jean-Marie Lagniez, Pierre Marquis, and Armin Biere. Dynamic blocked clause elimination for projected model counting. In *Theory and Applications of Satisfiability Testing (SAT)*, 2024.

33 Fangzhen Lin and Ray Reiter. Forget it! In *AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

34 Pierre Marquis. Knowledge compilation using theory prime implicates. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 837–843, 1995.

35 Sybylle Möhle and Armin Biere. Dualizing projected model counting. In *IEEE Conference on Tools with Artificial Intelligence*, pages 702–709, 2018.

36 Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, volume 7310 of *LNCS*, pages 356–361, 2012.

37 David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

38 Steven Prestwich. CNF encodings. In *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009.

39 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1169–1176, 2019.

40 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*. Springer, 2008.

41 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified progagation redundancy and compositional UNSAT checking in CakeML. *International Journal of Software Tools for Technology Transfer*, 25:167–184, 2023.

42 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

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