Proofs for Propositional Model Counting

Johannes K. Fichte
TU Wien, Austria

Markus Hecher
TU Wien, Austria

Valentin Roland
secunet Security Networks AG, Essen, Germany

Abstract

Although propositional model counting (#SAT) was long considered too hard to be practical, today's highly efficient solvers facilitate applications in probabilistic reasoning, reliability estimation, quantitative design space exploration, and more. The current trend of solvers growing more capable every year is likely to continue as a diverse range of algorithms are explored in the field. However, to establish model counters as reliable tools like SAT-solvers, correctness is as critical as speed.

As in the nature of complex systems, bugs emerge as soon as the tools are widely used. To identify and avoid bugs, explain decisions, and provide trustworthy results, we need verifiable results. In this paper, we propose a novel system for certifying model counts. We show how proof traces can be generated efficiently for exact model counters based on dynamic programming, counting CDCL with component caching, as well as knowledge compilation to Decision-DNNF. These approaches are the most widely used techniques in exact implementations. Furthermore, we provide proof-of-concept software for emitting proofs from model counters and a parallel trace checker. Based on this, we show the feasibility of using certified model counting in an empirical experiment.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Proof theory; Mathematics of computing → Combinatorics; Software and its engineering → Formal software verification

Keywords and phrases Model Counting, Verification, Counting Proofs

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Propositional model counting, also known as #SAT, asks to output the number of satisfying assignments of a propositional formula. The problem is canonical for the complexity class #P [54, 46, 1] and by consequence of Toda's theorem [53] a polynomial-time machine can solve with a single call to a #SAT oracle any problem in the Polynomial Hierarchy. While #SAT was long considered impractical, the field has seen considerable advances in recent years and highly efficient solvers emerged, capable of solving larger problems each year [8, 41, 37, 38, 49, 17, 21]. The applications of these solvers are vast. In artificial intelligence and reasoning, model counting is key when using logic-based reasoning for symbolic quantitative tasks [13, 3]. Model counters are becoming a standard tool for answering quantitative queries on propositional theories, in domains like configuration analysis [51], probabilistic reasoning [1, 39], explainable artificial intelligence [50, 2], or risk analysis [56, 18].

In fields such as explainability, risk analysis, or verification, we need to be able to trust the output of a model counter. Similar to SAT solvers, model counters are highly efficient, but complex software systems making it hard to trust their outputs. As in any complex system [25], subtle implementation errors may easily remain undiscovered. In fact, in the course of this research, we *spotted such an error* in the model counter sharpSAT [52], which also serves as a basis for several other implementations, like Ganak [49] and Dsharp [40]: If preprocessing is disabled, the solver does not propagate unit clauses present in the input

© Johannes Fichte, Markus Hecher, Valentin Roland;
licensed under Creative Commons License CC-BY 4.0

formula. In some cases, this leads to illegal assignments to their variables and wrong model counts. This highlights that even a widely used solver can hide implementation errors, especially in lesser-used code paths.

▶ **Example 1** (Spotted Bug in sharpSAT). Consider a formula consisting of two unit clauses $F = (a) \wedge (b)$. It is easy to see that this formula has exactly one model. However, without preprocessing, sharpSAT produces a model count of 4. Since input unit clauses are not propagated, a and b are considered unconstrained during component analysis.

Since verifying entire complex solvers, which are constantly enhanced by algorithmic innovations, is currently effectively impossible, researchers came up with a way to certify outputs from SAT solvers [4, 5, 33]. The idea is simple but extremely effective. The solver emits a trace during solving in a specific proof system focusing on simple constructs. In that way, decisions made by a solver can be externally verified. The tool to prove correctness of the trace can itself be fully verified and remains stable – even if the solver changes. While one might not be able to verify all techniques in the solver, one can ensure that every step is correct by verifying simple mathematical properties on the trace. This led to incredible stability improvements in the solvers, in particular for complex parts such as inprocessing [33]. However, propositional proof systems are tailored towards unsatisfiability [9] and there is no obvious way to encode a count in a sequence of existing clausal proofs [28], as enumerating models quickly becomes infeasible. So far no generic proof system for model counting exists.

In model counting, we are interested in properties that are between satisfiability and equivalence of the formulas. Mathematical combinatorics and number theory already provides us with basic tools to establish counting proofs [29]. One possible principle is *double-counting*, which checks whether two different approaches return the same answer. However, while the diversity of algorithms for model counting makes correlated errors in two different solvers less likely, there may still be a common conceptual error. In addition, double-counting limits us to instances for which more than one solver returns a count. Another classical principle is to establish a *proof by bijection*. Two sets are shown to have the same number of members by exhibiting a bijection, i.e., a one-to-one correspondence, between them. We employ the principle of proofs by bijection and establish a system that is tailored to practical model counting. To this end, we formalize systematic search space splitting, which is a common technique in model counting, and employ existing clausal proofs for unsatisfiability.

Contributions. Our main contributions are as follows:

1. We propose a novel proof system for certifying propositional model counts in practice.
2. We show how proof traces can be generated efficiently for exact model counters based on dynamic programming, counting version of CDCL with component caching, and knowledge compilation to Decision-DNNF.
3. We provide proof-of-concept software for emitting proofs from model counters and a parallel trace checker. Based on this, we show the feasibility of using certified model counting in an empirical experiment.

Related Works. Over time, various proof systems were developed for certifying SAT solver outputs [26, 33, 55, 10], among them the popular format DRAT [55]. These approaches aim to show unsatisfiability as a satisfying assignment can anyways be checked efficiently. While there is no general method for verifying exact model counts, outputs of knowledge compilers can be validated by equivalence checking [6, 7]. One approach is to label unsatisfiable subformulas, during knowledge compilation, with a clause indicating the cause of the conflict [6].

This allows checking equivalence to a formula in conjunctive normal form (CNF) in polynomial time under restricted conditions. Since it conflicts with component caching employed in knowledge compilers like D4 [38] and DSHARP [40], more recent works introduce a more flexible notion of syntactical equivalence [7]. While knowledge compilers, counting version of CDCL-based algorithms, and dynamic programming-based solving techniques are related [32], the two latter techniques are not accommodated in the certification approach from above. For example, the best performing solver of the 2021 Model Counting Competition did not use knowledge compilations [36]. Furthermore, the notion of equivalence used for certifying knowledge compilations is stronger than needed for preserving correctness of counting. In consequence, we expect that certain counting-specific simplifications cannot be formulated within this framework. Model counting on Decision-DNNFs can easily be done in terms of complexity [14], but technically the result is not verified. Hence, errors in the counting step will not be caught by checking equivalence to the input formula. Finally, we would like to mention that our focus is on exact model counting. Some modern counting techniques rely on probabilistic exact counting or approximate counting, which is incredibly valuable for scalable counting of very large instances in applications where the exact count is of less relevance. For approximate counting, the idea is to reduce the solution space uniformly to a small number of samples. By varying the number and length of randomly chosen XOR constraints and the number of repetitions, approximate counting can produce arbitrarily tight bounds with arbitrarily high confidence. Here, our approach is not meaningful, already existing techniques for certifying XOR constraints can be used [43, 27] instead.

2 Preliminaries

We assume that the reader is familiar with basic notions on functions, set theory, computational complexity [42], and propositional logic [35].

Propositional Satisfiability and Model Counting (#SAT). A *literal* ℓ is a propositional variable v or its negation $\neg v$. Conversely, we refer to the variable v of literals v and $\neg v$ by $\mathrm{var}(v) := v$ and $\mathrm{var}(\neg v) := v$, respectively. We assume $\neg\neg v$ to be equivalent to writing v and, for a set L of literals, let $\neg L := \{\neg v \mid v \in L\}$. A finite set C of literals is called a *clause*, interpreted as a disjunction of literals. We say that clause C is *unit* if $|C| = 1$. We refer by $\mathrm{vars}(C)$ to the variables occurring in a finite set C of literals, i.e., $\mathrm{vars}(C) := \{\mathrm{var}(\ell) \mid \ell \in C\}$. A *formula* F in conjunctive normal form (CNF) is a finite set of clauses, interpreted as a conjunction of clauses. Unless otherwise stated, we assume that formulas are in CNF. We let $\mathrm{vars}(F)$ denote the variables occurring in a formula, i.e., $\mathrm{vars}(F) := \bigcup_{C\in F} \mathrm{vars}(C)$. An *assumption* is a set A of literals. We let the unit clauses $\hat{A}$ for an assumption A be the formula $\hat{A} := \{\{\ell\} \mid \ell \in A\}$, i.e., the formula enforcing assumption A. A *restriction* of a set of literals A to set V of variables is $A|_V := A \cap (V \cup \neg V)$. Such a restriction can be applied to assumptions and clauses. A *restriction* of a formula F to set V of variables is $\{C|_V \mid C \in F\}$, i.e., the formula with each of the clauses in F restricted to V. A (partial) *assignment* is a function $\alpha : V \to \{0, 1\}$ which maps variables from a set V to values $\{0, 1\}$. For the negation of a variable $v \in V$, we define $\alpha(\neg v) := 1 - \alpha(v)$. By 2^V we denote the set of all total assignments $\alpha : V \to \{0, 1\}$, by $\alpha^{-1}(b)$ the preimage of α for $b \in \{0, 1\}$, and by $\mathrm{dom}(\alpha) := V$ its domain. A *restriction* $\alpha|_V$ of an assignment α to a set V of variables is the partial assignment that assigns variables $v \in V \cap \mathrm{dom}(\alpha)$ to $\alpha(v)$ and is undefined otherwise. For a formula F, we let $F[\alpha] := \{C \setminus \alpha^{-1}(0) \mid C \in F, C \cap \alpha^{-1}(1) = \emptyset\}$ be the formula F under assignment α. An assignment α *falsifies* F if $\emptyset \in F[\alpha]$; α *satisfies* F if $F[\alpha] = \emptyset$. *Note that assignment α can falsify F, satisfy F, or neither (unless α is total).*

Later, the definitions are used *precisely* as stated for α over *any* set V of variables. Further, α satisfies a clause C if $\{C\}$ is satisfied. An assignment α is called a *model* of a formula F if $\text{dom}(\alpha) = \text{vars}(F)$ and α satisfies F. Note that for an assumption A, we can satisfy $\hat{A}$ trivially by assignment τ_A, where for a variable $v \in \mathbb{V}$, $\tau_A(v) := 1$ if $v \in A$ and $\tau_A(v) := 0$ if $\neg v \in A$. We let $\mathsf{Mod}(F)$ be the set $\{\alpha \mid \alpha \in 2^V, \alpha \text{ satisfies } F, V = \text{vars}(F)\}$, i.e., the set of all models of a formula F over variables $\text{vars}(F)$. Then, $|\mathsf{Mod}(F)|$ is the *model count* of F. While the SAT problem asks whether $|\mathsf{Mod}(F)| \geq 1$, the *model counting problem*, or #SAT for short, asks to output $|\mathsf{Mod}(F)|$. In addition, for a tuple $\mathcal{T} = (F, V)$ consisting of formula F and set V of variables and assumption A, we let $\mathsf{Mod}_A(\mathcal{T}) := \{\alpha \mid \alpha \in 2^V, \alpha \text{ satisfies } F \cup \hat{A}\}$ be the *models of $\mathcal{T}$ under assumption* A. Intuitively, tuple $\mathcal{T} = (F, V)$ represents formula F restricted to variables V if $V \subset \text{vars}(F)$ or adding unconstrained variables to F if $V \supset \text{vars}(F)$. Then, the *model count* for $\mathcal{T}$ under assumption A is $|\mathsf{Mod}_A(\mathcal{T})|$. Finally, a formula F *entails* a clause C, written as $F \models C$, if all models of F satisfy C.

Certified SAT Solving. Certified results are common in SAT solving [26, 30, 24] and support of a standardized format is mandatory for solvers taking part in the competition [24]. Popular formalisms like DRAT [30] and RUP are examples of the more general notion of *propositional proof systems* [9]. Let Σ be an alphabet, $L \subseteq \Sigma^*$ be language, and $\texttt{TAUT}$ be the class of all propositional formulas that are tautological, encoded in a fixed alphabet. In general, a *proof system* is a polynomial-time computable function $s : \Sigma_1^* \to \Sigma^*$ with range L that maps from words over a proof alphabet Σ_1 to words in L. For a formula F, if $s(x) = F$ then $x \in \Sigma_1$ is called a *proof* in system s. A *propositional proof system* is a proof system for $\texttt{TAUT}$. Usual properties are *completeness* asking that every propositional tautology has a proof in system s and *soundness* asking that if a propositional formula has a proof in system s then it is a tautology. Further, proofs need to be verifiable in polynomial time of their size. Intuitively, propositional proof systems concern *certificates* of membership in $\texttt{TAUT}$ for a given formula. Since SAT solving usually works on CNF formulas as input, practical focus is on clausal proofs. *Clausal proofs* are sequences of clauses, where each clause is entailed by the original formula [28]. A clausal proof is called a *refutation*, or *proof of unsatisfiability*, if it contains the empty clause. The input formula is unsatisfiable if the empty clause is derived in this sequence. Proof variants are based on clauses that have the RUP (reverse unit propagation) [26] and RAT (resolution asymmetric tautology) property. These proof formats share verifiability in polynomial time in the size of the proof and input formula and can be tightly coupled with modern solving techniques. The popular DRAT proof system uses a more general redundancy property based on extended resolution instead of entailment [30].

3 From Clausal Proofs Towards Certifying #SAT

By basic constructions, we can use propositional proof systems to establish correctness of an outputted model count. To this end, we systematically enumerate models and add the negation of a found model to the formula, thereby forbidding it. After solving outputs that all models were found, we can prove that no more models exist by using clausal proofs of unsatisfiability. Alternatively, we can establish equivalence using propositional proof systems when solving by means of knowledge compilation. In both cases we use decision-based proof systems for an actual function problem. Clearly, propositional proof systems lack the capability of reasoning about sets of models and their cardinality, instead, we can only reason about individual yes/no decisions. It is easy to see that *enumerating models quickly becomes infeasible* and an equivalence-based approach works only for very specific techniques. Moreover, a decision-based approach is not how modern model counters reason. Instead,

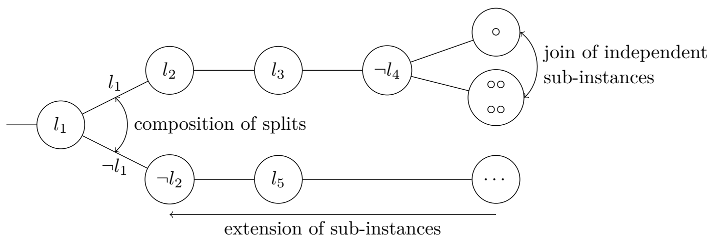


Figure 1 Visual representation of operations used by #SAT solvers when exploring the solution space. Splits in the search space are composed, independent sub-instances are joined and sub-instances are extended by assigning new variables.

they commonly split the search space into sub-problems where solutions can be considered independently, so-called *components*, as illustrated on a high level in the example below.

▶ **Example 2** (Refutations are Insufficient for Counting)**.** Consider formula $F = (a \vee b) \wedge (c \vee d)$, which has 9 models. In detail, there are 3 assignments to a and b that satisfy $(a \vee b)$ and 3 assignments to c and d to satisfy $(c \vee d)$. Since both clauses share no variables, we can freely combine the satisfying assignments for each clause. Hence, we obtain $3 \cdot 3 = 9$ models.

Example 2 states a reasoning technique that cannot be expressed concisely in a clausal proof. In fact, we miss notions to express the combination of both sub-problems and lack the capability of reasoning about sets of models and their cardinalities. Consequently, we are interested in a natural approach to obtain certifiably correct results for practical propositional model counters. From a more theoretical perspective, we design a proof system for counting where the computable function has range $\mathbb{N}_0$. The challenging part is to define a system that is simple, but expressive enough that certificates can be generated by diverse model counting algorithms with low overhead. Mathematical combinatorics and number theory already provides us with basic tools to establish counting proofs [29]. A classical concept in proofs is to establish relations between sets using bijections, i.e., a one-to-one correspondence. Two sets have the same number of elements if there is a bijection between them.

▶ **Example 3.** Consider again Example 2. We can easily show that there is a bijective mapping between the set obtained from the Cartesian product of models of both clauses and the set of models of the formula F, i.e., $\mathsf{Mod}(F) = \mathsf{Mod}(a \vee b) \times \mathsf{Mod}(c \vee d)$. Hence, the model count of formula F can simply be expressed as the product of the cardinality of both sets, i.e., $|\mathsf{Mod}(F)| = |\mathsf{Mod}(a \vee b)| \cdot |\mathsf{Mod}(c \vee d)|$.

Below, we formalize systematic search space splitting, which is common among all exact model counting approaches. Then, we establish general reasoning rules based on this technique and we combine it with established techniques on clausal proofs for `TAUT`. The overall principle remains. Solvers output certificates, which can be easily verified by a program (*checker*). Correctness of the checker needs to be audited for fully verified results.

3.1 Search Space Splitting

State-of-the-art model counters split the input instance into sub-instances and combine the results – even implicitly in knowledge compilation-based or dynamic programming-based counters. Figure 1 visualizes solution space exploration and operations that modern solvers use to combine solutions of sub-instances. Intuitively, we observe the following principles: (i) Solvers split the search space along an assumption, e.g., a decision literal. The corresponding reasoning operation is a *composition* of disjoint sets of models for a sub-instance. (ii) Solvers identify sub-instances that are independent or overlap in such a way that their models can be combined in a database-like *join*. (iii) Under some restrictions, models of a sub-instance can be *extended* to models of a larger one by adding a stricter assumption. (iv) There are sub-instances that do not have any models.

Next, we introduce components as a formal notion for sub-instances and claims as statements talking about the model count of components under an assumption. Intuitively, a component consists of a subset of the input formula and a selection of variables.

▶ **Definition 4** (Component and Sub-components)**.** *A* component $\mathcal{C} = (F, V)$ *consists of a formula F and a set V of variables. We say that a component $\mathcal{C}_s = (F_s, V_s)$ is a* sub-component *of $\mathcal{C}$ if $F_s \subseteq F$ and $V_s \subseteq V$.*

The set of models of a component can be restricted by an *assumption*, i.e., a set of unit literals. Then, models can be represented by stating the component and assumption, which we can directly use for counting. We call such a statement a *claim*, which represents a partial result. The following example illustrates sets of models for components under assumptions.

▶ **Example 5.** Consider formula $F = (a \vee b) \wedge (c \vee d)$ from Example 3 and corresponding component $\mathcal{C} = (F, \text{vars}(F))$. Recall that without an assumption, F has 9 models. If we add assumption $A_1 = \{a, \neg b\}$, then $\mathsf{Mod}_{A_1}(\mathcal{C}) = \{\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 0\}, \{a \mapsto 1, b \mapsto 0, c \mapsto 0, d \mapsto 1\}, \{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1\}\}$, hence $|\mathsf{Mod}_{A_1}(\mathcal{C})| = 3$. With assumption $A_2 = \{\neg a, \neg b\}$ we have $\mathsf{Mod}_{A_2}(\mathcal{C}) = \emptyset$.

▶ **Definition 6** (Claim)**.** *Let $\mathcal{C} = (F, V)$ be a component. A* claim $\mathcal{I} = (\mathcal{C}, A, c)$ *over $\mathcal{C}$ consists of an assumption A with* $\text{vars}(A) \subseteq V$ *and a count $c \in \mathbb{N}_0$. $\mathcal{I}$ is* correct *if $c = |\mathsf{Mod}_A(\mathcal{C})|$.*

Working with claims over components avoids naive enumeration of models, which we can see in the following example.

▶ **Example 7.** Consider a formula F with a large number of clauses and variables. Further, we have two disjoint components $\mathcal{C}_1 = (F_1, V_1)$ and $\mathcal{C}_2 = (F_2, V_2)$ in formula F (i.e., $F_1 \cap F_2 = \emptyset$ and $V_1 \cap V_2 = \emptyset$). Using claims, we succinctly express that any two of the models can be combined into a model of a larger component $\mathcal{C} = (F_1 \cup F_2, V_1 \cup V_2)$: If $(\mathcal{C}_1, \emptyset, c_1)$ and $(\mathcal{C}_2, \emptyset, c_2)$ are correct claims, then $(\mathcal{C}, \emptyset, c_1 \cdot c_2)$ is correct, regardless of c_1 and c_2. In contrast, enumeration would quickly become infeasible in practice, since the individual components may have exponentially many models.

3.2 (Re-)combining Claims of Search Spaces

As already demonstrated by the previous example, we can model search space splitting of modern model counters by the combination of claims. In general, however, it is not guaranteed that we do not over- or undercount when combining claims. Hence, we consider properties of sets of claims that dictate which claims can be combined correctly.

Avoid Overcounting. First, we demonstrate a case where overcounting occurs.

▶ **Example 8.** Let $\mathcal{C} = (\{\{a, b, c\}, \{a, \neg d\}\}, \{a, b, c, d\})$ be a component $(\mathcal{C}, \{a, b\}, 4)$ and $(\mathcal{C}, \{a, d\}, 4)$ be claims of $\mathcal{C}$. Then, both claims count the model $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$, i.e., they overlap and cannot be simply combined without overcounting.

Consequently, we consider non-overlapping sets of claims, which prevents overcounting.

▶ **Definition 9** (Non-overlapping and Uniform Claims)**.** *Let $\mathcal{C} = (F, V)$ be a component and S be a set of claims over $\mathcal{C}$.*

- *Then, S is* non-overlapping *if, for every two distinct claims $(\mathcal{C}, A_1, c_1)$ and $(\mathcal{C}, A_2, c_2)$ in set S, we have $\mathsf{Mod}_{A_1}(\mathcal{C}) \cap \mathsf{Mod}_{A_2}(\mathcal{C}) = \emptyset$.*
- *Let $U \subseteq V$. If $\mathrm{vars}(A') = U$ for every $(\mathcal{C}, A', c') \in S$, we call S* uniform *for U.*

Observe that uniformness is a special case of non-overlapping that is easier to verify.

▶ **Observation 10** ($\star$[1]). *Let $\mathcal{C} = (F, V)$ be a component, $U \subseteq V$ a set of variables, and S a set of claims over $\mathcal{C}$. If S is uniform for U and all claims in S are correct, S is non-overlapping.*

Provably Prevent Undercounting. While non-overlapping sets of claims prevent overcounting, we also need to ensure the absence of further models to prevent undercounting. Before we show how to avoid this, we briefly illustrate undercounting below.

▶ **Example 11.** Consider a component $\mathcal{C} = (\{\{a, b, c\}, \{a, \neg d\}\}, \{a, b, c, d\})$ and claims $\mathcal{I}_1 = (\mathcal{C}, \{a, d\}, 4)$, $\mathcal{I}_2 = (\mathcal{C}, \{a, \neg d\}, 4)$ of $\mathcal{C}$. $\mathcal{I}_1$ and $\mathcal{I}_2$ are non-overlapping, but cannot correctly be combined to $\mathcal{I} = (\mathcal{C}, \emptyset, 8)$, because models with $a \mapsto 0$ are counted by neither $\mathcal{I}_1$ nor $\mathcal{I}_2$. However, adding a third claim $(\mathcal{C}, \{\neg a, d\}, 3)$ is sufficient to cover $\mathcal{C}$ exhaustively, since there are no models with $a \mapsto 0$ and $d \mapsto 1$.

To prove that undercounting does not occur, we need to ensure that a set of claims covers the models of a component *exhaustively*. Note that this is different from a single claim with zero models, which states that its component is unsatisfiable under an assumption. Rather, exhaustiveness can be seen as unsatisfiability with exceptions.

▶ **Definition 12** (Exhaustive Claims). *Let $\mathcal{C} = (F, V)$ be a component and S a set of claims over $\mathcal{C}$. For an assumption A, we call S* exhaustive *for A if for every model $\alpha \in \mathsf{Mod}_A(\mathcal{C})$ there is a claim $(\mathcal{C}, A', c') \in S$ with α satisfies $\hat{A}'$ and $A \subseteq A'$.*

Unfortunately, already the task of checking exhaustiveness is hard.

▶ **Proposition 13** ($\star$, Exhaustiveness of Claims is co-NP Hard). *Let S be a set of claims for a component $\mathcal{C} = (F, V)$ that is uniform for $U \subseteq V$ and A be an assumption with $\mathrm{vars}(A) \subseteq U$. Then, it is co-NP-complete to decide whether S is exhaustive for A.*

Despite this result, we later show that exhaustiveness can be established efficiently for the surveyed solver implementations. This is not a contradiction, since #SAT is known to be in $\#P$ [54], and thus at least as hard as decision problems in NP and co-NP. Hence, when using intermediate results from the solving process for exhaustiveness checking, the "hardness" lies in computing this intermediate information. However, if a set of claims is uniform, we can show exhaustiveness, using well-known constructs from SAT. Next, we create a shortcut for exhaustiveness of sets of claims, formalized in the notion of an *absence of models statement*.

▶ **Definition 14** (Absence of Models Statement). *Let $\mathcal{C} = (F, V)$ be a component, S be a set of claims that is uniform for $U \subseteq V$, and A be an assumption where we have $A \subseteq A'$ for every claim $(\mathcal{C}, A', c) \in S$. Then, given a clausal proof Δ, we call $\mathcal{A} = (\mathcal{C}, A, U, \Delta)$ an* absence of models statement. *Such a statement $\mathcal{A}$ is* correct *if Δ is a refutation of $\hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$.*

Intuitively absence of models can be seen as acting complementarily to claims. To state that a component under some assumption is unsatisfiable, except for a set of claims, we employ the well-established concepts of unsatisfiability proofs in SAT solving. Indeed this is sufficient for exhaustiveness.

[1] Proofs of statements marked with $\star$ will be made public in an extended version.

▶ **Lemma 15** (⋆,Absence of Models). *Let $\mathcal{A} = (\mathcal{C}, A, U, \Delta)$ be an absence of models statement and S be a set of claims for component $\mathcal{C}$ that is uniform over U and we have $A \subseteq A'$ for every claim $(\mathcal{C}, A', c) \in S$. If $\mathcal{A}$ is correct for S, then S is exhaustive for A.*

Non-overlapping and Exhaustive Claims. To avoid *both over- and undercounting*, we combine the properties non-overlapping and exhaustive. We say that a non-overlapping set of claims that is exhaustive for A is *composable to A*.

▶ **Example 16.** Consider a component $\mathcal{C} = (\{\{a, b, c\}, \{a, \neg d\}\}, \{a, b, c, d\})$ and a set S of claims of $\mathcal{C}$ with assumptions $\{a, d\}$, $\{a, \neg d\}$, and $\{\neg a, \neg d\}$. Since the assumptions are distinct and S is uniform for $U = \{a, d\}$, S is non-overlapping. Since $\mathcal{C}$ has no model with $a \mapsto 0$ and $d \mapsto 1$, S is exhaustive for $\emptyset$. To prove this, we can easily find a refutation Δ for $\{\{\dots\}, \{a, \neg d\}\} \cup \{\{\neg a, \neg d\}, \{\neg a, d\}, \{a, d\}\}$. Hence, we can construct a correct absence of models statement $(\mathcal{C}, \emptyset, U, \Delta)$ and know that S is composable to $\emptyset$.

Note that we can restrict a set of claims that is uniform for U and is composable to some assumption, to a subset with a stricter assumption. This subset remains composable to that stricter assumption. Hence, a single absence of models statement can be used to show composability for multiple subsets of such a set of claims.

▶ **Observation 17** (Composable Subset). *Let S be a set of claims for a component $\mathcal{C} = (F, V)$, that is composable to A and uniform for U. Let A_s be an assumption with $A \subset A_s$ and $\mathrm{vars}(A_s) \subseteq U$. Then, $S_{A_s} := \{(\mathcal{C}, A', c') \in S \mid A_s \subseteq A'\}$ is composable to A_s.*

3.3 Inference Rules for Model Counting

To reproduce the reasoning of a solver, we use a trace, which is a finite sequence of steps. Each step is either a claim, which represents a set of models, or an absence of models statement.

▶ **Definition 18** (Model Counting Trace). *A* model counting trace *$T = \langle s_1, \dots, s_n \rangle$ for a given formula F is a finite sequence of steps s_i, which is either a claim or an absence of model statement. A trace T is* complete *for F if there is a claim $\mathcal{I} = (\mathcal{C}, \emptyset, c)$ in T for a component $\mathcal{C} = (F, \mathrm{vars}(F))$. A trace T is* correct *if all claims in T are correct.*

Next, we establish inference rules, which allow us to mechanically verify correctness of a combination of partial results, which are succinctly expressed by claims. Since we approach model counting proofs from the perspective of capturing search space splitting performed by a solver instead of a formal deductive system, our rules use syntactic as well as semantic notions. While it is possible to express them purely in syntactical terms as inference rules of a deductive system, we defer this to future work. To make this clear, we denote a rule as premises and inference separated by a *double rule*.

Intuitively, these rules then establish the correctness of a claim by combining claims of sub-components or claims with stronger assumptions. In other words, we bring the combination of claims in line with splitting the search space by composition, join, and extension, as visualized in Figure 1. The resulting basic inference rules are *(exactly one) model*, *composition*, *join*, and *extension*. Composition itself is backed up by exhaustiveness, thereby using the sufficient absence of models statement via clausal proofs, cf., Lemma 15.

Inferring Exactly One Model Claims. First, we cover the base case of exactly one model, claiming exactly one model for a component under a (total) assumption A. There, no further claim is needed to check its correctness. Indeed, since a model is a total assignment over the variables of the given component (F, V), it suffices to check whether the formula F of the component is satisfied by assumption A. As a result, we obtain the following simple rule.

$$\frac{\tau_A \text{ satisfies } F}{((F, \mathrm{dom}(\tau_A)), A, 1)}$$

▶ **Lemma 19** (⋆,Exactly One Model)**.** *Let $\mathcal{C} = (F, V)$ be a component and $(\mathcal{C}, A, 1)$ be a claim for $\mathcal{C}$ with $\mathrm{vars}(A) = V$. The claim is correct if and only if τ_A satisfies F.*

Inferring Composition Claims. Next, we discuss the composition inference rule. If we have a set S of correct claims of a component that is composable to an assumption A, we can directly infer a claim with the more general assumption A. Intuitively, we derive a more general statement from a set of more specific claims. A matching absence of models statement represents proof that S meets the necessary conditions to avoid undercounting, i.e., that S is exhaustive for A. However, by Observation 10 such a statement also implies that S is non-overlapping, thereby avoiding overcounting as well. Therefore, the following rule assumes composability of every claim $(\mathcal{C}, A_i, c_i)$ in S as a requirement.

$$\frac{(\mathcal{C}, A_1, c_1), \quad \ldots \quad, (\mathcal{C}, A_n, c_n) \quad \text{are correct and composable to } A}{(\mathcal{C}, A, \Sigma_{1 \leq i \leq n} c_i)}$$

▶ **Lemma 20** (⋆,Composition or Unsatisfiability)**.** *Let $\mathcal{C} = (F, V)$ be a component, $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim with assumption A, S be a set of claims over $\mathcal{C}$ that is composable to A, and $c := \sum_{(\mathcal{C}, A', c') \in S} c'$. If every claim in S is correct, then $\mathcal{I}$ is correct, i.e., $|\mathsf{Mod}_A(\mathcal{C})| = c$.*

Note that if $S = \emptyset$, composition states that the current component is *unsatisfiable under assumption* A. Extending Example 16, the following example illustrates inference by composition of a composable set of claims.

▶ **Example 21.** Consider a component $\mathcal{C} = (F, V)$ with $F = \{\{a, b, c\}, \{a, \neg d\}\}$ and $V = \{a, b, c, d\}$. Recall from Example 16, that a set S of claims with assumptions $\{a, d\}$, $\{a, \neg d\}$, and $\{\neg a, \neg d\}$, is composable to $\emptyset$. By composition, we infer claim $(\mathcal{C}, \emptyset, 11)$, as shown below.

$$\text{set } S: \left.\begin{array}{ll} c & A \\ \hline 4 & \{\ a \ \ d\ \} \\ 4 & \{\ a \ \neg d\ \} \\ 3 & \{\ \neg a \ \neg d\ \} \end{array}\right\} \longrightarrow \text{composed claim: } \begin{array}{ll} c & A \\ \hline 11 & \emptyset \end{array}$$

In addition, we know that $\emptyset$ is composable to $A = \{\neg a, d\}$. By composition, we infer a claim $(\mathcal{C}, A, 0)$. This is equivalent to stating that $\mathcal{C}$ is unsatisfiable under assumption A.

Inferring Join Claims. If the models of two components are independent, we can combine them arbitrarily to models of a joint component. The join rule generalizes this idea allowing models to overlap, thereby assuming correct claims $((F_1, V_1), A_1, c_1)$ and $((F_2, V_2), A_2, c_2)$.

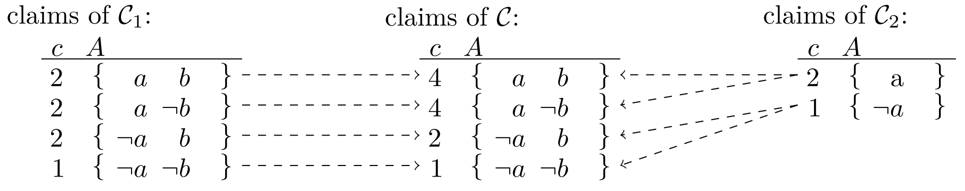
$$\frac{\begin{array}{l} ((F_1, V_1), A_1, c_1) \text{ and } ((F_2, V_2), A_2, c_2) \text{ are correct} \\ A_1 \cup A_2 \text{ is consistent and } V_1 \cap V_2 \subseteq \mathrm{vars}(A_1 \cup A_2) \\ \forall i \in \{1, 2\}, C \in F_i : \mathrm{vars}(C) \cap (V_1 \cup V_2 \setminus V_i) = \emptyset \end{array}}{((F_1 \cup F_2, V_1 \cup V_2), A_1 \cup A_2, c_1 \cdot c_2)}$$

In case of overlapping components, the variables shared by the joined components must be constrained by the inferred assumption and the clauses of one component must not further constrain the set of models of the other.

▶ **Lemma 22** (⋆,Join). *Let $\mathcal{C} = (F, V)$ be a component; $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim; $\mathcal{C}_1 = (F_1, V_1)$ and $\mathcal{C}_2 = (F_2, V_2)$ be sub-components of $\mathcal{C}$ with $F = F_1 \cup F_2$, $V = V_1 \cup V_2$, and $V_1 \cap V_2 \subseteq \text{vars}(A)$ where every $C \in F_i$ has $\text{vars}(C) \cap (V \setminus V_i) = \emptyset$. If $\mathcal{I}_1 = (\mathcal{C}_1, A|_{V_1}, c_1)$ and $\mathcal{I}_2 = (\mathcal{C}_2, A|_{V_2}, c_2)$ are correct claims over $\mathcal{C}_1$ and $\mathcal{C}_2$ and $c = c_1 \cdot c_2$, then $\mathcal{I}$ is correct, i.e., $|\mathsf{Mod}_A(\mathcal{C})| = c$.*

Note that if the model count of either joined claim is zero, the joint count is zero, regardless of the other claim. The following example illustrates joins with overlapping assumptions.

▶ **Example 23.** Consider components $\mathcal{C}_1 = (F_1, V_1)$ and $\mathcal{C}_2 = (F_2, V_2)$ with $F_1 = \{\{a, b, c\}\}$, $V_1 = \{a, b, c\}$, $F_2 = \{\{a, \neg d\}\}$, and $V_2 = \{a, d\}$. The tables below represent claims for both components, along with claims for a third component $\mathcal{C} = (F_1 \cup F_2, V_1 \cup V_2)$. All claims of $\mathcal{C}$ are inferred from claims of sub-components $\mathcal{C}_1$ and $\mathcal{C}_2$ using Lemma 22.



claims of $\mathcal{C}_1$:

c	A
2	$\{\ a\ \ b\ \}$
2	$\{\ a\ \neg b\ \}$
2	$\{\ \neg a\ \ b\ \}$
1	$\{\ \neg a\ \neg b\ \}$

claims of $\mathcal{C}$:

c	A
4	$\{\ a\ \ b\ \}$
4	$\{\ a\ \neg b\ \}$
2	$\{\ \neg a\ \ b\ \}$
1	$\{\ \neg a\ \neg b\ \}$

claims of $\mathcal{C}_2$:

c	A
2	{ a }
1	$\{\ \neg a\ \}$

Extension. Similar to extending models to assign additional variables, we can extend a claim to a larger component by adding additional literals to its assumption. We formalize this in the following rule, whereby we assume a correct claim $((F', V'), A', c)$.

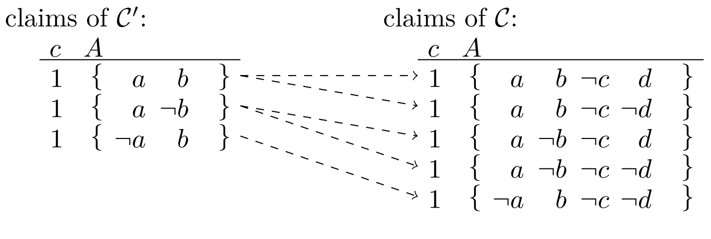
$$\frac{\begin{array}{l} ((F', V'), A', c) \text{ is correct} \\ \tau_A \text{ satisfies } F \setminus F', \text{ where } F \supseteq F' \text{ and } A|_{V'} = A' \\ \forall C \in F' : \tau_A|_{V \setminus V'} \text{ does not satisfy } C \end{array}}{\text{((F,V), A, c)}}$$

Since we infer a claim with the same count as the extended claim, introduced variables must be constrained by the extended assumption, models extended according to the assumption must satisfy the larger component, and no additional models may be introduced.

▶ **Lemma 24** (⋆,Extension). *Let $\mathcal{C} = (F, V)$ be a component, $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim, $\mathcal{C}' = (F', V')$ be a sub-component of $\mathcal{C}$, and $\mathcal{I}' = (\mathcal{C}', A|_{V'}, c)$ be a correct claim. If $V \setminus V' \subseteq \text{vars}(A)$, τ_A satisfies $F \setminus F'$, and $\tau_A|_{V \setminus V'}$ does not satisfy C for every clause $C \in F'$, then $\mathcal{I}$ is correct.*

In principle, one might ask why we cannot add arbitrary literals to the assumption when extending a claim. However, every model of a claim $\mathcal{I}'$ requires a corresponding model in the extended claim $\mathcal{I}$. We ensure this by enforcing that clauses that are only in the larger component $\mathcal{C}$, are satisfied by the literals added to the assumption. Conversely, every model of the claim $\mathcal{I}$, when restricted to the variables of $\mathcal{C}'$, must be a model of $\mathcal{I}'$. Intuitively, this establishes a sufficient one-to-one correspondence between the models of $\mathcal{I}$ and $\mathcal{I}'$.

▶ **Example 25.** We extend a component $\mathcal{C}' = (F', V')$ with $F' = \{\{a, b, c\}\}$ and $V' = \{a, b\}$ to the component $\mathcal{C} = (F, V)$ with $F = \{\{a, b, c\}, \{\{a, \neg d\}\}$ and $V = \{a, b, c, d\}$. The tables below list claims for both components, where the claims of $\mathcal{C}$ can be verified by Lemma 24 (extension). Note that the extended assumption must include $\neg c$ to satisfy extension.



claims of $\mathcal{C}'$:

c	A
1	$\{\ a\ \ b\ \}$
1	$\{\ a\ \neg b\ \}$
1	$\{\ \neg a\ \ b\ \}$

claims of $\mathcal{C}$:

c	A
1	$\{\ a\ \ b\ \neg c\ \ d\ \}$
1	$\{\ a\ \ b\ \neg c\ \neg d\ \}$
1	$\{\ a\ \neg b\ \neg c\ \ d\ \}$
1	$\{\ a\ \neg b\ \neg c\ \neg d\ \}$
1	$\{\ \neg a\ \ b\ \neg c\ \neg d\ \}$

▶ Remark 26. Let $\mathcal{C} = (\emptyset, \emptyset)$ be a component. Since $\mathsf{Mod}_\emptyset(\mathcal{C}) = \{\emptyset\}$, claim $\mathcal{I} = (\mathcal{C}, \emptyset, 1)$ is correct. Thus, a claim satisfying exactly one model can be expressed as an extension of $\mathcal{I}$.

3.4 Proofs for Model Counting

In the previous section, we established principles for reasoning on correctness of claims based on claims obtained during search space splitting. These principles allow us to construct proofs for #SAT. Specifically, we aim for traces that are proofs for model counting where each step can be inferred from the preceding steps. To this end, we introduce MICE steps that employ the inference rules as given in Section 3.1. We also state the corresponding correctness lemma for convenience in brackets.

▶ **Definition 27** (MICE Proof Step). *Let $T = \langle s_1, \ldots, s_n \rangle$ be a model counting trace for a given formula F and $S = \{s_1, \ldots, s_n\}$. We say $s_r \notin S$ is a* Model-counting Induction by Claim Extension (MICE) *step from S if either condition below is satisfied:*

- *s_r is an absence of models statement $(\mathcal{C}, A, U, \Delta)$ that is correct for $\{(\mathcal{C}, A', c') \mid (\mathcal{C}, A', c') \in S, A \subseteq A', \mathrm{vars}(A') = U\}$ (cf., Lemma 15);*
- *s_r is a claim of exactly one model; (cf., Lemma 19);*
- *s_r is a claim joining two claims $\mathcal{I}_1, \mathcal{I}_2 \in S$ (cf., Lemma 22);*
- *s_r is a claim extending another claim $\mathcal{I} \in S$ (cf., Lemma 24); or*
- *s_r is a claim $(\mathcal{C}, A, c)$ by composing a set $S' \subseteq S$ of claims and there is an absence of model statement $(\mathcal{C}, A, U, \Delta)$ correct for S', hence S' is composable to A (cf., Lemma 20).*

This leads to our central definition of when a model counting trace is actually a MICE proof.

▶ **Definition 28** (MICE Proofs). *Let $T = \langle s_1, \ldots, s_n \rangle$ be a model counting trace that is complete for a given formula F. If every s_i in T is a MICE step from $\{s_1, \ldots, s_{i-1}\}$, then T is a* MICE proof.

Indeed, MICE proofs are sound, i.e, suitable for proving the model count of a given formula. Furthermore, when restricting model counting to MICE proofs we do not loose completeness, i.e., a MICE proof exists for any formula.

▶ **Theorem 29** ($\star$,Soundness & Completeness). *Given formula F, component $\mathcal{C}{=}(F, \mathrm{vars}(F))$.*

- Soundness*: If T is a MICE proof, then F has $c{=}|\mathsf{Mod}(F)|$ many models.*
- Completeness*: There exists a MICE proof T that is complete for F.*

4 Verifying Model Counting Traces

Having established model counting traces, we design an algorithm that can verify whether a trace is correct and complete. Further, we demonstrate that model counting traces can be verified *efficiently*, i.e., we can check whether such a trace is a MICE proof in polynomial time in the size of the trace and the input formula. By Theorem 29, MICE proofs are sufficient. In the following, we discuss a simple polynomial-time algorithm for checking correctness of counting traces. We focus on correctness, because completeness can easily be checked by linearly searching a trace for a relevant claim.

Consider Algorithm 1, which takes as input a trace T and outputs either “Correct” or “Incorrect”. For each step s_i in $T = \langle s_1, \ldots, s_n \rangle$, we check if s_i is a MICE step from its predecessors $P = \{s_1, \ldots, s_{i-1}\}$ by testing each case in Definition 27 sequentially. If no case applies to s_i, we terminate with “Incorrect”. After processing all steps in T we output “Correct”. Algorithm 1 outputs “Correct” if and only if the trace is a MICE proof.

Algorithm 1 Simple Trace Correctness Checking

```
Input: A model counting trace $T = \langle s_1, \dots, s_n \rangle$
Output: "Correct" if $T$ is a MICE proof, "Incorrect" otherwise
for $i \in \{1, \dots, n\}$ do
    $P := \{s_1, \dots, s_{i-1}\}$                                   ▷ Set of predecessors.
    if $s_i$ is an absence of models statement $(\mathcal{C}, A, U, \Delta)$ then
        $S := \{(\mathcal{C}, A_j, c) \in S \mid A_j \supseteq A, \mathrm{vars}(A_j) = U\}$
        $F_E := \hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$
        if $\Delta$ is refutation of $F_E$ then continue
    else if $s_i$ is a claim $(\mathcal{C}, A, c)$ with $\mathcal{C} = (F, V)$ then
        if $\mathrm{vars}(A) = V$ and $c = 1$ and $\tau_A$ satisfies $F$ then continue    ▷ Lemma 19
        else if $s_i$ is correct by joining $s_j, s_k \in P$ then continue              ▷ Lemma 22
        else if $s_i$ is correct by extending $s_j \in P$ then continue                ▷ Lemma 24
        else if $P$ contains some $(\mathcal{C}, A, U, \Delta)$ then
            $S := \{(\mathcal{C}, A_j, c) \in S \mid A_j \supseteq A, \mathrm{vars}(A_j) = U\}$
            if $s_i$ is correct by composition of $S$ then continue                   ▷ Lemma 20
    return Incorrect                                                 ▷ Step verification failed.
return Correct
```

▶ **Proposition 30** (⋆,Polynomial-Time Correctness Checking). *Given a model counting trace T, Algorithm 1 runs in polynomial time in the size of T.*

5 Practical Considerations and Preliminary Evaluation

In this section, we briefly explain practical improvements, solver integration, and provide preliminary data on using our certificates for model counting. We implemented Algorithm 1 into a program, which we call SHARPCHECK that is *open-source and available* at `https://github.com/vroland/sharptrace`. For space reasons, we describe the improvements such as identifiers for components allowing us to restrict the search for claims to a single component in an extended version. If we assume that for one assumption, there exists only one claim in a component with that assumption, claims can be verified in parallel in any order. Additionally, we can ensure that a refutation for an assumption A' has to be checked only once by allowing to reference an absence of models step in a more general assumption A, which follows directly from Observation 17.

5.1 Solver Integration

For practical use, we integrate MICE traces into existing techniques for exact model counting. Our approach works for generating traces in exact model counting when using a counting version of CDCL with component caching [48], dynamic programming [47], or knowledge compilation [14, 38]. For each technique, we provide a conceptual description and an implementation. We directly augmented two solvers with tracing capabilities and compile Decision-DNNFs outputted by knowledge compilers into MICE traces. Our implementations are proof-of-concepts relying on existing SAT solvers for generating clausal proofs.

sharpSAT The solver uses a counting version of CDCL with component caching for model counting [52]. It serves as basis for several model counters, e.g., Dsharp [40] or GANAK [49].

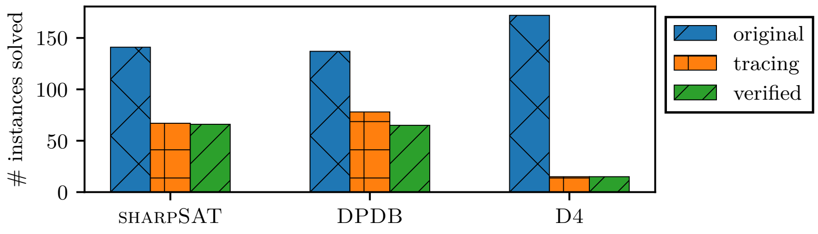


Figure 2 Number of solved instances with and without tracing for sharpSAT, DPDB, and D4. Of the instances solved with tracing, we show the number of traces verified given the same timeout.

The implementation emits traces directly during the solving process. Refutations for absence of models are efficiently extracted during conflict analysis. Our implementation is available online at `https://github.com/vroland/sharpSAT/tree/proof-trace`.

DPDB The solver implements dynamic programming algorithms on TDs using database management systems [23]. We extended the model counting implementation to translate result tables into claims and use miniSAT [19] to generate refutations. Our implementation is available online at `https://github.com/vroland/dp_on_dbs/tree/sharpsat_proof`.

D4 For prototype on knowledge compilation, we generate traces from the Decision-DNNF while using miniSAT for refutations. We consider D4 due to its performance in a past competition [21]. We believe that modifying D4 would result in faster trace outputs, but is out of scope for our prototypical considerations. However, we provide conceptual details for modifying a knowledge compiler in an extended version.
Our implementation is available online at `https://github.com/vroland/nnf2trace`.

5.2 Empirical Evaluation

To demonstrate the capability of our approach and estimate the overhead of traces in practice, we conducted a preliminary evaluation on 400 instances of varying hardness and size.

Design of Experiment. We draw a small empirical experiment to study the following questions: (Q1) Is there significant impact when solving with traces by compared solving without emitting traces? (Q2) Can we verify traces in a reasonable time? (Q3) Does the technique help to find bugs?

Instances. We considered sets of instances from Tracks 1 and 2 of the Model Counting Competition 2020 [21]. For Track 2, we removed the weights. Instances are available on a public data repository [22].

Hardware, Measure, and Restrictions. All solvers run on a server with two physical Intel Xeon Silver 4112CPUs, where each of these 16 runs at 2.60GHz and has access to 128GB RAM. Results are gathered on Ubuntu 18.04 LTS powered on kernel 4.15.0-135 with hyperthreading disabled. We allow a solving time of 600 seconds per instance. Since we do not implement traces directly into D4, we apply timeout to the combined runtime of D4 and the trace generation. We execute solvers sequentially, one at a time, limiting available memory to the maximum available on the system. Checking of the traces runs in parallel on 16 cores with a timeout of 600 seconds per instance.

Limitations. All implementations are prototypical and are not optimized towards efficiency. Traces currently do not support pre-processing.

Solver	$\#_O$	$\#_T$	$\#_V$
sharpSAT	141	67	66
DPDB	137	78	65
D4	172	15	15

(a) Number of solved instances.

Solver	#	$t_O[h]$	$t_T[h]$	$t_{T+V}[h]$	mem$[GB]$
sharpSAT	66	0:24:19	2:56:01	3:24:14	233
DPDB	65	0:23:27	2:44:45	1:29:28	17
D4	15	0:01:13	0:23:28	0:00:46	1

(b) Runtimes for verified instances and total proof size.

Table 1 Performance of solving and trace checking. $\#_.$ refers to number of instances, t the total runtime, mem memory, O without tracing, T with tracing, and T+V with tracing and checking.

Performance of Solving with Trace Outputs

Towards answering the question on the impact of traces on solving time, we consider the number of solved instances with and without proof logging within the considered timeout. We survey the number of solved instances in Figure 2 and provide more details in Table 1. sharpSAT and DPDB solved a similar number of instances when outputting traces. However, when comparing the number of solved instances with and without trace output, outputting traces reduced the number of instances to half. For D4, we could generate traces only for a small number of instances in the available time. In Table 1b, we list the size of proofs. For sharpSAT, we can see that proofs grow quite large.

Discussion. To explain the results in more detail, we consider each solver individually. For sharpSAT, most overhead in generating trace outputs is introduced by I/O functions and serializing the output, which can be determined from profiling the solver. Currently, component definitions are only implicitly stored and have to be re-constructed. Refutations can directly be constructed within sharpSAT. For DPDB, fetching large intermediate results from the database and passing them to miniSAT to generate refutations causes significant overhead. For Decision-DNNFs, we currently need to construct a high number of refutations, which requires to call an external solver resulting in a notable overhead. We believe that this can be done directly inside a knowledge compiler. In comparison to DPDB, there is a much higher number of refutations, however, refutations are of much smaller size.

Summary. Overall, the results show that we are capable of generating traces with common solving techniques in practice. However, emitting proof traces results in significantly lower performance of the solvers. In fact, within the same timeouts, only half of the instances could be solved. Generating traces from Decision-DNNFs is not yet suitable as generating refutations for absence of models suffer notable overhead, which might be already outputted during solving. Still, our prototypes hopefully serve as a basis for future implementations.

Performance of Verifying Traces

Next, we consider the question of whether traces can be verified in a reasonable time. Therefore, we consider the number of verified instances in comparison to the number of solved instances with traces logging within the considered timeout. Figure 2 illustrates an overview of the number of verified traces by sharpCheck and Table 1 provides additional details. For sharpSAT, all but one trace could be verified within the timeout. The remaining trace takes ≈ 750 seconds to verify on our system. Similarly, a large portion of DPDB traces could be verified. The traces generated from Decision-DNNFs were all verified.

Discussion. When verifying traces from sharpSAT, most time is spent on parsing and validating the integrity of the trace, while claims and refutations are checked very quickly. In contrast, the traces emitted by DPDB are more compact and most time is spent on verifying refutations. In dynamic programming algorithms, large sets of claims are composed resulting in large refutations in absence of model steps. When traces have more than 10^6 refutation steps in total, checking often timed out. We expect improvement from adding deletion information to refutations, like in DRAT.

Summary. When comparing accumulated runtimes for solving and checking in Table 1b, we see that checking MICE traces is not only efficient in theory, but also in practice. Still, we expect that checking performance can be further improved when certified counting matures.

Finding Bugs

During our experiments, we noticed a bug in sharpSAT, which we already outlined in Example 1. If preprocessing is disabled, unit clauses are violated and the solver outputs a wrong model count. The MICE trace allowed us to pinpoint the actual origin of the issue. Furthermore, we discovered that outputting traces may interfere with the two watched literals scheme used in sharpSAT. This resulted in wrong counts for some benchmark instances. We located and fixed the bug using MICE traces. Despite these bugs, which show up only under certain conditions, we did not discover major issues. This confirms common mathematical intuition that double counting already improves correctness. In addition, it supports observations made in the Model Counting Competition 2020 that current model counters are quite robust. However, the situation might look different as soon as preprocessing is included and if only one model counter gives a solution similar to SAT solving [34, 31].

6 Conclusion and Future Work

Model counters are key tools for symbolic quantitative reasoning. Exact model counters need to be trustworthy, in particular, in fields such as explainability, risk analysis, or verification. While proof logging and verification approaches exist for SAT, a common proof system for exact model counting was missing. Previous approaches to correctness were either limited to a specific counting algorithm by establishing equivalence or could only be used to show correctness of steps in approximate counting. In this paper, we propose a novel approach to certified #SAT based on traces that capture the solution space exploration during solving. We show that clausal proofs used for certifying unsatisfiability in SAT solvers are insufficient for #SAT. Instead, we propose a system for certifying outputs from propositional model counters practice, where we use clausal proofs as basic building blocks. We demonstrate that our approach can be applied to solvers based on CDCL variants with component caching, dynamic programming on tree decompositions, and knowledge compilation to Decision-DNNFs. We provide prototypes for each solving technique and a tool for automated trace checking. Finally, we illustrate preliminary results for certified model counting in actual solvers.

Our work opens up a wide variety of directions. A prime candidate for future investigations is an efficient integration into knowledge compilers and dynamic programming-based solving that uses more sophisticated data structures [16]. Further, establishing more general, but efficiently verifiable, inference rules may facilitate integration into solvers. Here, stronger proof techniques might come in handy [20, 27]. Beyond simple model counting, extending counting traces to weighted model counting or projected model counting, which is highly relevant in practical applications, seems to be a natural step for future considerations. Finally, although our implementation to verify traces is conceptually simple, an efficient, formally verified implementation might be interesting for highly sensitive applications.

References

1 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proceedings of the 44th Symposium on Foundations of Computer Science*, FOCS'03, pages 340–351, Cambridge, MA, USA, 2003. IEEE Computer Soc. doi:10.1109/SFCS.2003.1238208.

2 Teodora Baluta, Zheng Leong Chua, Kuldeep S. Meel, and Prateek Saxena. Scalable quantitative verification for deep neural networks. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, ICSE'21, pages 312–323, Madrid, 2021. IEEE Computer Soc. doi:10.1109/ICSE43902.2021.00039.

3 Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. Pseudo polynomial-time top-k algorithms for d-DNNF circuits, 2022. arXiv:2202.05938.

4 Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT'09, pages 1–5, Montreal, Canada, 2009. Association for Computing Machinery, New York. doi:10.1145/1670412.1670413.

5 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In Ofer Strichman and Stefan Szeider, editors, *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT'10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57, Edinburgh, UK, July 2010. Springer Verlag. doi:10.1007/978-3-642-14186-7_6.

6 Florent Capelli. Knowledge compilation languages as proof systems. In Mikolás Janota and Inês Lynce, editors, *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing*, volume 11628 of *Lecture Notes in Computer Science*, pages 90–99, Lisbon, 2019. Springer Verlag. doi:10.1007/978-3-030-24258-9_6.

7 Florent Capelli, Jean-Marie Lagniez, and Pierre Marquis. Certifying top-down decision-dnnf compilers. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, pages 6244–6253, Virtual Event, 2021. The AAAI Press.

8 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Improving approximate counting for probabilistic inference: From linear to logarithmic SAT solver calls. In Subbarao Kambhampati, editor, *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 3569–3576, New York City, NY, USA, July 2016. The AAAI Press.

9 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symbolic Logic*, 44(1):36–50, 1979. doi:10.2307/2273702.

10 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Proceedings of the 26th International Conference on Automated Deduction (CADE'17)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236, Gothenburg, Sweden, 2017. Springer Verlag. doi:10.1007/978-3-319-63046-5_14.

11 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics*, 11(1-2):11–34, 2001. doi:10.3166/jancl.11.11-34.

12 Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, ECAI'04, pages 328–332, Valencia, Spain, 2004. IOS Press.

13 Adnan Darwiche. Three modern roles for logic in AI. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20)*, pages 229–243, New York, NY, USA, 2020. Association for Computing Machinery, New York. doi:10.1145/3375395.3389131.

14 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. doi:10.1613/jair.989.

15 Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate Texts in Mathematics*. Springer Verlag, 2012.

16 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI'20)*, pages 1468–1476, New York, 2020. The AAAI Press.

17 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. Dpmc: Weighted model counting by dynamic programming on project-join trees. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, pages 211–230, Louvain-la-Neuve, Belgium, September 2020. Springer Verlag.

18 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, pages 4488–4494, San Francisco, California, USA, 2017. The AAAI Press.

19 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer Verlag. `doi:10.1007/978-3-540-24605-3_37`.

20 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-boolean reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1486–1494, Apr. 2020. `doi:10.1609/aaai.v34i02.5507`.

21 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *ACM Journal of Experimental Algorithmics*, 26(13), December 2021. `doi:10.1145/3459080`.

22 Johannes K. Fichte, Markus Hecher, and Valentin Roland. GPUSAT3 benchmark data and source code, 2021. `doi:10.5281/zenodo.5159903`.

23 Johannes Klaus Fichte, Markus Hecher, Patrick Thier, and Stefan Woltran. Exploiting database management systems and treewidth for counting. In Ekaterina Komendantskaya and Yanhong Annie Liu, editors, *Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages*, volume 12007 of *Lecture Notes in Computer Science*, pages 151–167, New Orleans, LA, USA, 2020. Springer Verlag. `doi:10.1007/978-3-030-39197-3_10`.

24 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. `doi:10.1016/j.artint.2021.103572`.

25 Daniel E. Geer Jr. Complexity is the enemy. *IEEE Security Privacy*, 6(6):88–88, 2008. `doi:10.1109/MSP.2008.139`.

26 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *Proceedings of the International Symposium on Artificial Intelligence and Mathematics 2009*, ISAIM'08, Fort Lauderdale, Florida, USA, 2008.

27 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. In Kevin Leyton-Brown and Mausam, editors, *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, pages 3768–3777, 2021.

28 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE'03)*, pages 10886–10891, Munich, Germany, 2003. IEEE Computer Soc. `doi:10.1109/DATE.2003.10008`.

29 Ronald Lewis Graham, Martin Grötschel, and László Lovász. *Handbook of combinatorics*, volume I. Elsevier Science Publishers, North-Holland, 1995.

30 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (CADE'24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359, Lake Placid, NY, USA, 2013. Springer Verlag. `doi:10.1007/978-3-642-38574-2_24`.

31 Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the 25th*

International Conference on Automated Deduction Automated Deduction (CADE'25), pages 591–606. Springer Verlag, 2015.

32 Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 156–162, Edinburgh, Scotland, UK, 2005. Professional Book Center.

33 Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370, Manchester, UK, June 2012. Springer Verlag. `doi:10.1007/978-3-642-31365-3_28`.

34 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR'12)*, pages 355–370. Springer Verlag, 2012.

35 Hans Kleine Büning and Theodor Lettman. *Propositional Logic: Deduction and Algorithms*, volume 48 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1999.

36 Tuukka Korhonen and Matti Järvisalo. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP'21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:11, Dagstuhl, Germany, 2021. Dagstuhl Publishing. `doi:10.4230/LIPIcs.CP.2021.8`.

37 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 751–757, New York City, NY, USA, July 2016. The AAAI Press.

38 Jean-Marie Lagniez and Pierre Marquis. An improved Decision-DDNF compiler. In Carles Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 667–673, Melbourne, VIC, Australia, 2017. The AAAI Press.

39 Anna L. D. Latour, Behrouz Babaki, Anton Dries, Angelika Kimmig, Guy Van den Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and probabilistic programming. In J. Christopher Beck, editor, *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP'17)*, pages 495–511. Springer Verlag, 2017. `doi:10.1007/978-3-319-66158-2_32`.

40 Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In Leila Kosseim and Diana Inkpen, editors, *Proceedings of the 25th Canadian Conference on Artificial Intelligence*, volume 7310 of *Lecture Notes in Computer Science*, pages 356–361, Toronto, ON, Canada, 2012. Springer Verlag. `doi:10.1007/978-3-642-30353-1_36`.

41 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 3141–3148, Buenos Aires, Argentina, 2015. The AAAI Press.

42 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

43 Tobias Philipp and Adrián Rebola-Pardo. DRAT proofs for XOR reasoning. In Loizos Michael and Antonis Kakas, editors, *Proceedings of 15th European Conference on Logics in Artificial Intelligence (JELIA'16)*, pages 415–429. Springer Verlag, 2016. `doi:10.1007/978-3-319-48758-8_27`.

44 Alan Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*. Elsevier Science Publishers, North-Holland, 2001.

45 John Alan Robinson. A machine-oriented logic based on the resolution principle. *Association for Computing Machinery, New York*, 12(1):23–41, 1965. `doi:10.1145/321250.321253`.

46 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996. `doi:10.1016/0004-3702(94)00092-1`.

47 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'07)*, volume 4790 of *Lecture Notes in Computer Science*, pages 484–498, Yerevan, Armenia, 2007. Springer Verlag. `doi:10.1007/978-3-540-75560-9_35`.

48 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

49 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, pages 1169–1176, Macao, China, 2019. IJCAI.

50 Weijia Shi, Andy Shih, Adnan Darwiche, and Arthur Choi. On tractable representations of binary neural networks. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR'20)*, pages 882–892, Rhodes, Greece, 2020. `doi:10.24963/kr.2020/91`.

51 Chico Sundermann, Thomas Thüm, and Ina Schaefer. Evaluating #SAT solvers on industrial feature models. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS '20)*, New York, NY, USA, 2020. Association for Computing Machinery, New York. `doi:10.1145/3377024.3377025`.

52 Marc Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429, Seattle, WA, USA, 2006. Springer Verlag. `doi:10.1007/11814948_38`.

53 Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *Society for Industrial and Applied Mathematics (SIAM)*, 20(5):865–877, 1991. `doi:10.1137/0220053`.

54 Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979. `doi:10.1016/0304-3975(79)90044-6`.

55 Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Proceedings of the 17th International Conference Theory and Applications of Satisfiability Testing (SAT'14)*, pages 422–429, Vienna, Austria, July 2014. Springer Verlag. Held as Part of the Vienna Summer of Logic, VSL 2014. `doi:10.1007/978-3-319-09284-3_31`.

56 Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In Ranjita Bhagwan and George Porter, editors, *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 575–589, Santa Clara, CA, USA, 2020. USENIX Association.

A Omitted Proofs

(Re-)combining Claims of Search Spaces

▶ **Observation 10.** *Let $\mathcal{C} = (F, V)$ be a component, $U \subseteq V$ a set of variables, and S a set of claims over $\mathcal{C}$. If S is uniform for U and all claims in S are correct, S is non-overlapping.*

Proof. Recall that since S is uniform for U, we have $\mathrm{vars}(A) = U$ for all claims $(\mathcal{C}, A, c)$ in S. Then, for every assignment α that is total over V, there is exactly one assumption A with $\mathrm{vars}(A) = U$ and α satisfies $\hat{A}$. Assume all claims in S are correct. Since there is one correct count for a component and assumption, no two claims in S have the same assumption. Hence, for every total assignment α to V, there is exactly one matching assumption over variables U, thus there is at most one claim $(\mathcal{C}, A, c)$ in S with $\alpha \in \mathsf{Mod}_A(\mathcal{C})$. ◀

▶ **Proposition 13** (Exhaustiveness of Claims is co-NP Hard). *Let S be a set of claims for a component $\mathcal{C} = (F, V)$ that is uniform for $U \subseteq V$ and A be an assumption with $\mathrm{vars}(A) \subseteq U$. Then, it is co-NP-complete to decide whether S is exhaustive for A.*

Proof. We show co-NP-completeness of deciding exhaustiveness of a set of claims S for an assumption A by polynomial-time reduction from (hardness) and to (completeness) UNSAT, which is a co-NP-complete problem. UNSAT asks to decide whether a given formula is unsatisfiable, i.e., it is the complement of SAT.

To show hardness, we provide a polynomial-time reduction from UNSAT: Let r be a function that maps a formula F to an instance of the exhaustiveness problem that asks whether the set $S = \emptyset$ is exhaustive for component $\mathcal{C} = (F, \mathrm{vars}(F))$ and assumption $\emptyset$. We know that F is in UNSAT if and only if $\mathsf{Mod}_\emptyset(\mathcal{C}) = \emptyset$. By Definition 12, that is the case if and only if $S = \emptyset$ is exhaustive for the assumption $\emptyset$. Hence, F is in UNSAT if and only if $S = \emptyset$ is exhaustive for $\emptyset$. Since $r(F)$ can be computed in polynomial time in the size of a formula F, r is a reduction from UNSAT to the exhaustiveness problem.

For showing membership in co-NP and thus co-NP-completeness, let $\mathcal{C} = (F, V)$ be a component, $U \subseteq V$ a set of variables, and A an assumption with $\mathrm{vars}(A) \subseteq U$, and S a set of claims of component $\mathcal{C}$ with assumptions over variables U. Then, a function r that outputs the formula $F_E := \hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$ from S, $\mathcal{C}$, and A is a reduction to UNSAT by Lemma 15: From Definition 14 and Lemma 15 follows that if there is a refutation for F_E, i.e. F_E is unsatisfiable, S is exhaustive for A.

Finally, we show that r is computable in polynomial time in the size of S, $\mathcal{C}$, and A. It is easy to see that the inputs to r can be encoded as strings of symbols and comparison and inversion of literals can be performed in polynomial time. We first output $\hat{A}$ from A. Then, we output $\{C|_V \mid C \in F\}$, where we need $\mathcal{O}(|C| \cdot |V|)$ steps per clause $C \in F$. Finally, we output $\neg A'$ for every $(\mathcal{C}, A', c') \in S$, taking polynomial time in the size of S. Hence, the function r is a reduction to UNSAT and computable in polynomial time.

Since UNSAT can be reduced to exhaustiveness of a uniform set of claims and vice versa in polynomial time, the exhaustiveness problem is co-NP complete. ◀

▶ **Lemma 15** (Absence of Models). *Let $\mathcal{A} = (\mathcal{C}, A, U, \Delta)$ be an absence of models statement and S be a set of claims for component $\mathcal{C}$ that is uniform over U and we have $A \subseteq A'$ for every claim $(\mathcal{C}, A', c) \in S$. If $\mathcal{A}$ is correct for S, then S is exhaustive for A.*

Proof. Let $\mathcal{C} = (F, V)$. Recall that $\mathcal{A}$ is correct if Δ is a refutation of $E = \hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$. For proof of contradiction, assume S is not exhaustive for A. Then, there is a model $\alpha \in \mathsf{Mod}_A(\mathcal{C})$, but there is no claim with assumption A_e in S, such that $\tau_{A_e} = \alpha|_U$. Since $\alpha \in \mathsf{Mod}_A(\mathcal{C})$, assignment α satisfies $\hat{A} \cup \{C|_V \mid C \in F\}$. Hence, assignment α must falsify $\{\neg A' \mid (\mathcal{C}, A', c') \in S\}$, or else E is satisfiable and Δ is not a refutation. However, since no claim with A_e is in S, α satisfies $\neg A'$ for all assumptions A' of claims in S by construction. Thus, α satisfies E, which contradicts that Δ be a refutation. Hence, if Δ is a refutation for E and hence $\mathcal{A}$ is correct, S is exhaustive for A. ◀

Inference Rules and Proof Traces for Model Counting

▶ **Lemma 19** (Exactly One Model)**.** *Let $\mathcal{C} = (F, V)$ be a component and $(\mathcal{C}, A, 1)$ be a claim for $\mathcal{C}$ with $\mathsf{vars}(A) = V$. The claim is correct if and only if τ_A satisfies F.*

Proof. Consider the assignment $\alpha = \tau_A$. Since $\mathsf{vars}(A) = V$, assignment α is total over V and $\mathsf{Mod}_A(\mathcal{C}) \subseteq \{\alpha\}$. Consequently, τ_A satisfies F if and only if we have $\mathsf{Mod}_A(\mathcal{C}) = \{\alpha\}$ and $|\mathsf{Mod}_A(\mathcal{C})| = 1$. ◀

▶ **Lemma 20** (Composition or Unsatisfiability)**.** *Let $\mathcal{C} = (F, V)$ be a component, $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim with assumption A, S be a set of claims over $\mathcal{C}$ that is composable to A, and $c := \sum_{(\mathcal{C}, A', c') \in S} c'$. If every claim in S is correct, then $\mathcal{I}$ is correct, i.e., $|\mathsf{Mod}_A(\mathcal{C})| = c$.*

Proof. Since S is composable to A, it is exhaustive for A and non-overlapping. Further, by exhaustiveness for A, for all models α in $\mathsf{Mod}_A(\mathcal{C})$, there is a claim $(\mathcal{C}, A', c)$ in S with $\alpha \in \mathsf{Mod}_{A'}(\mathcal{C})$. Hence, $\mathsf{Mod}_A(\mathcal{C}) = \bigcup_{(\mathcal{C}, A', c') \in S} \mathsf{Mod}_{A'}(\mathcal{C})$. Since S is also non-overlapping, the assignments in $\mathsf{Mod}_{A'}(\mathcal{C})$ are mutually disjoint. Thus, if every claim in S is correct, we have $|\mathsf{Mod}_A(\mathcal{C})| = \sum_{(\mathcal{C}, A', c') \in S} c' = c$. ◀

▶ **Lemma 22** (Join)**.** *Let $\mathcal{C} = (F, V)$ be a component; $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim; $\mathcal{C}_1 = (F_1, V_1)$ and $\mathcal{C}_2 = (F_2, V_2)$ be sub-components of $\mathcal{C}$ with $F = F_1 \cup F_2$, $V = V_1 \cup V_2$, and $V_1 \cap V_2 \subseteq \mathsf{vars}(A)$ where every $C \in F_i$ has $\mathsf{vars}(C) \cap (V \setminus V_i) = \emptyset$. If $\mathcal{I}_1 = (\mathcal{C}_1, A|_{V_1}, c_1)$ and $\mathcal{I}_2 = (\mathcal{C}_2, A|_{V_2}, c_2)$ are correct claims over $\mathcal{C}_1$ and $\mathcal{C}_2$ and $c = c_1 \cdot c_2$, then $\mathcal{I}$ is correct, i.e., $|\mathsf{Mod}_A(\mathcal{C})| = c$.*

Proof. We distinguish two cases.

Case (I), $c_1 = 0$ or $c_2 = 0$. Then, $\mathsf{Mod}_{A|_{V_1}}(\mathcal{C}_1) = \emptyset$ or $\mathsf{Mod}_{A|_{V_2}}(\mathcal{C}_2) = \emptyset$, since the sub-component claims are correct. Assume $c_1 = 0$. Then, for every possible assignment over V_1, a clause $C \in F_1$ exists that is not satisfied. But, because $\mathsf{vars}(C) \cap (V \setminus V_1) = \emptyset$, C cannot be satisfied by an assignment over $V \setminus V_1$. Thus, C cannot be satisfied by any assignment over V_2. Hence, $|\mathsf{Mod}_A(\mathcal{C})| = 0 = c$. If $c_2 = 0$, the proof works similar.

Case (II), $c_1 \neq 0$ and $c_2 \neq 0$. We will show that $f : \alpha \mapsto (\alpha|_{V_1}, \alpha|_{V_2})$ is a bijection from $\mathsf{Mod}_A(\mathcal{C})$ into the set $J = \mathsf{Mod}_{A|_{V_1}}(\mathcal{C}_1) \times \mathsf{Mod}_{A|_{V_2}}(\mathcal{C}_2)$, similarly as in related work [47]. Since $|J| = \left|\mathsf{Mod}_{A|_{V_1}}(\mathcal{C}_1)\right| \cdot \left|\mathsf{Mod}_{A|_{V_2}}(\mathcal{C}_2)\right| = c_1 \cdot c_2$, we then conclude that $|\mathsf{Mod}_A(\mathcal{C})| = c_1 \cdot c_2$.

First, we establish that f is a mapping from $\mathsf{Mod}_A(\mathcal{C})$ to J. For any $\alpha \in \mathsf{Mod}_A(\mathcal{C})$, $\alpha|_{V_1}$ is in $\mathsf{Mod}_{A|_{V_1}}(\mathcal{C}_1)$ because clauses F_1 cannot contain literals of variables that are in $V \setminus V_1$. Hence, clauses in F_1 must be satisfied by literal assignments in α that are also in $\alpha|_{V_1}$. By the same reasoning, we have $\alpha|_{V_2} \in \mathsf{Mod}_{A_2}(\mathcal{C}_2)$.

Further, f is injective, because for every $\alpha_i, \alpha_j \in \mathsf{Mod}_A(\mathcal{C})$ with $f(\alpha_i) = f(\alpha_j)$, we have that $\alpha_i|_{V_1} = \alpha_j|_{V_1}$ and $\alpha_i|_{V_2} = \alpha_j|_{V_2}$. Then, since $V_1 \cup V_2 = V$, $\alpha_i = \alpha_j$. To see that f is surjective, let $(\alpha_1, \alpha_2) \in J$. Then, the assignment $\alpha = \alpha_1 \cup \alpha_2$ is consistent, since $V_1 \cap V_2 \subseteq \mathsf{vars}(A)$ and A is consistent by definition. Since α_1 satisfies every clause F_1 α_2 satisfies every clause F_2, and α is consistent, α satisfies every clause in $F_1 \cup F_2 = F$. In consequence, $\alpha \in \mathsf{Mod}_A(\mathcal{C})$.

Finally, because the sub-component claims are correct, we have $|J| = c_1 \cdot c_2 = c$. This concludes the proof. ◀

▶ **Lemma 24** (Extension). *Let $\mathcal{C} = (F, V)$ be a component, $\mathcal{I} = (\mathcal{C}, A, c)$ be a claim, $\mathcal{C}' = (F', V')$ be a sub-component of $\mathcal{C}$, and $\mathcal{I}' = (\mathcal{C}', A|_{V'}, c)$ be a correct claim. If $V \setminus V' \subseteq \text{vars}(A)$, τ_A satisfies $F \setminus F'$, and $\tau_A|_{V\setminus V'}$ does not satisfy C for all clauses $C \in F'$, then $|\mathsf{Mod}_A(\mathcal{C})| = c$.*

Proof. We show that $|\mathsf{Mod}_{A|_{V'}}(\mathcal{C}')| = |\mathsf{Mod}_A(\mathcal{C})| = c$ by proving that the function $f : \alpha \mapsto \alpha|_{V'}$ is a bijective mapping from model $\mathsf{Mod}_A(\mathcal{C})$ of claim $\mathcal{I}$ to models $\mathsf{Mod}_{A|_{V'}}(\mathcal{C}')$ of $\mathcal{I}'$.

To see that f is a valid mapping, we show that if α is a model of $\mathcal{C}$, $\alpha|_{V'}$ also satisfies F'. For $\alpha|_{V'}$ to not satisfy F', there must be a clause $C \in F'$ that is satisfied by a literal l of a variable in $V \setminus V'$. Since we require that $V \setminus V' \subseteq \text{vars}(A)$, l is in $A|_{V\setminus V'}$. However, then $\tau_A|_{V\setminus V'}$ satisfies C, which is not allowed. Thus, we have $\alpha|_{V'} \in \mathsf{Mod}_{A|_{V'}}(\mathcal{C}')$ for all $\alpha \in \mathsf{Mod}_A(\mathcal{C})$ and f is a valid mapping.

To show that f is injective, consider $\alpha_i, \alpha_j \in \mathsf{Mod}_A(\mathcal{C})$, where $f(\alpha_i) = f(\alpha_j) = \alpha_i|_{V'} = \alpha_j|_{V'}$. Then, since there is exactly one assignment $\gamma = \tau_{A|_{V\setminus V'}}$ over variables $V \setminus V'$ where γ satisfies $\hat{A}$, we have $\alpha_i = \alpha_i|_{V'} \cup \gamma = \alpha_j|_{V'} \cup \gamma = \alpha_j$.

For surjectivity, let $\beta \in \mathsf{Mod}_{A|_{V'}}(\mathcal{C}')$ be a model of claim $\mathcal{I}'$. Because $V \setminus V' \subseteq \text{vars}(A)$, we can construct an assignment α as $\beta \cup \tau_A$. Then, β satisfies the clauses F', τ_A satisfies the clauses $F \setminus F'$, thus α is an assignment over the variables V that satisfies the clauses F. Hence, we have $\alpha \in \mathsf{Mod}_A(\mathcal{C})$.

Since f is bijective and $\mathcal{I}' = (\mathcal{C}', A|_{V'}, c)$ is correct, we have that $|\mathsf{Mod}_{A|_{V'}}(\mathcal{C}')| = |\mathsf{Mod}_A(\mathcal{C})| = c$. This concludes the proof. ◀

▶ **Observation 31** (Trace Correctness of *MICE* Steps). *Let $T = \langle s_1, \ldots, s_n \rangle$ be a model counting trace. If every s_i in T is a MICE step from $S = \{s_1, \ldots, s_{i-1}\}$, T is correct.*

Proof. We prove correctness of a trace T by induction over its steps.

Base case. The empty trace $\langle\rangle$ is correct by Definition 18.

Inductive step. Assume the sub-trace $T_{i-1} = \langle s_1, \ldots, s_{i-1} \rangle$ of T is correct and s_i is a MICE step from $S = \{s_1, \ldots, s_{i-1}\}$. If s_i is a claim that can be inferred by Lemma 19 (Exactly one Model), Lemma 22 (Join), or Lemma 24 (Extension), s_i is correct and, thus, $T_i = \langle s_1, \ldots, s_{i-1}, s_i \rangle$ is correct.

If s_i is an absence of models statement $(\mathcal{C}, A, U, \Delta)$ that is a MICE step from S, we know that there is a subset $S' = \{(\mathcal{C}, A', c') \in S \mid A \subseteq A', \text{vars}(A') = U\}$ of S that is composable to A by Lemma 15 (Absence of Models). Then, trace T_i is correct because trace T_{i-1} is correct and s_i is not a claim.

Finally, consider the case where s_i is a claim $(\mathcal{C}, A, c)$ and there is an absence of models step $\mathcal{A} = (\mathcal{C}, A, U, \Delta)$ in S. Since $\mathcal{A}$ is a MICE step from its predecessors, $\mathcal{A}$ is correct for some set $S' \subseteq S$. Then, s_i can be inferred as composition of S' by Lemma 20 (Composition). Thus, $T_i = \langle s_1, \ldots, s_{i-1}, s_i \rangle$ is correct. This concludes the proof. ◀

▶ **Theorem 29** (Soundness and Completeness). *Given formula F, component $\mathcal{C}{=}(F, \text{vars}(F))$.*

- Soundness*: If T is a MICE proof, then F has $c{=}|\mathsf{Mod}(F)|$ many models.*
- Completeness*: There exists a MICE proof T that is complete for F.*

Proof. First, we consider *soundness*: If all steps in T are MICE steps from the set of their predecessors in T, trace T is correct by Observation 31. Then, because T is correct and contains a claim $\mathcal{I} = (\mathcal{C}, \emptyset, c)$, the claim $\mathcal{I}$ is correct. Hence, $|\mathsf{Mod}_\emptyset(\mathcal{C})| = c = \mathsf{Mod}(F)$.

Next, we show *completeness*: Let $N := \mathsf{Mod}(F)$ be the set of models of formula F. By enumeration, it is easy to construct a, though impractically large, correct and complete counting trace as follows: First, we construct a set of claims $S := \{(\mathcal{C}, A_\alpha, 1) \mid \alpha \in N\}$

where $\tau_{A_\alpha} = \alpha$. Each claim in S is verifiable using Lemma 19, hence it is a MICE step from $\emptyset$. Since N contains all models of $\mathcal{C}$, there is no assignment that satisfies $E = \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$. Hence, there is a refutation Δ for E and a correct absence of models step $\mathcal{A} = (\mathcal{C}, A, V, \Delta)$. Such a refutation Δ must exist for every unsatisfiable formula E [45]. Since S is uniform for variables V, and exhaustive for $\emptyset$, it is composable to $\emptyset$. Hence, the claim $\mathcal{I} = (\mathcal{C}, \emptyset, c)$ where $c = |N|$ can be inferred by Lemma 20. Thus, $\mathcal{I}$ is a MICE step from claims S and absence of models statement $\mathcal{A}$.

Finally, we construct a trace $T = \langle \mathcal{I}_1, \ldots, \mathcal{I}_n, \mathcal{A}, \mathcal{I} \rangle$ with $\mathcal{I}_1, \ldots, \mathcal{I}_n \in S$. It is easy to see that T is complete for F since it contains $\mathcal{I}$. Since $\mathcal{I}_1, \ldots, \mathcal{I}_n$ are correct by MICE steps from $\emptyset$ and $\mathcal{I}$ is a MICE step from steps $\{\mathcal{I}_1, \ldots, \mathcal{I}_n, \mathcal{A}\}$, trace T is correct. This concludes the proof. ◀

Verifying Proof Traces

▶ **Proposition 30** (Polynomial-Time Correctness Checking). *Given a model counting proof trace T, Algorithm 1 runs in polynomial time in the size of T and F.*

Proof. We show that Algorithm 1 runs in polynomial time in the size the input trace T, we consider the processing time for a single step $s_i \in T$. Since we process each step s_i in T sequentially, if processing each step s_i takes polynomial time, Algorithm 1 runs in polynomial time. In this proof, we assume a naive sequential encoding of the steps of T, separated by separator symbols. For a step s_i, let the trace prefix P be the set of steps preceding s_i in T. Next, we each case in Definition 27.

If s_i is an absence of models statement $(\mathcal{C}, A, U, \Delta)$, we first construct a set of claims S from P. To construct S, we search through the trace prefix P and collect all claims for the current component with assumptions over U that are a subset of A. Then, we construct the formula $F_E := \hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (\mathcal{C}, A', c') \in S\}$. Both steps take polynomial time in the size of P and the size of F_E is bounded by the size of P. Hence, checking if Δ is a refutation of F_E takes at most polynomial time in the size of $P \subseteq T$.

In the case that s_i is a claim, we check the prerequisites for Lemma 19 (Exactly One Model), Lemma 22 (Join), and Lemma 24 (Extension) sequentially. It is easy to see that comparing sets of variables and clauses, calculating vars() for clauses and assumptions, verifying subset relations, and checking whether clauses are satisfied by an assignment are polynomial-time operations. Hence, we can check Lemma 19 efficiently. To check correctness by Lemma 24, we need to search the trace prefix P for a claim to infer s_i from. We can check Lemma 24 in polynomial time, and this search takes at most $|P| - 1$ checks. Analogously, searching for two claims to join by Lemma 22 takes at most $(|P| - 1)^2$ polynomial-time checks of Lemma 22.

Finally, we check correctness by Lemma 20 (Composition). We can find an absence of models step $(\mathcal{C}, A, U, \Delta)$ in $|P| - 1$ polynomial-time steps. Finding one such step is sufficient since using another absence of models statement must lead to the same inference, as all absence of models steps in P are correct. Constructing a set of claims $S \subseteq P$ as before and checking the claim count by Lemma 20 takes polynomial time.

Overall, checking if s_i is a MICE step from P case-by-case takes polynomial time in the size of P. Hence, Algorithm 1 checks correctness of a trace T in polynomial time. ◀

B Additional Plots

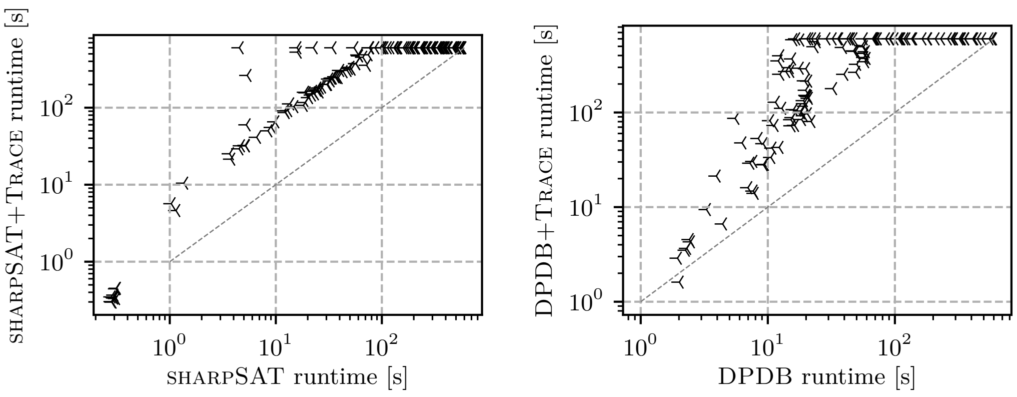


Figure 3 Scatter plot comparing runtimes of individual instances with and without tracing for sharpSAT (left) and DPDB (right).

C Solver Integration

For the sequel of this section, we assume that the reader is familiar with basic notions in graph theory and propositional logic such as entailment, otherwise we refer to standard texts [15, 35].

Knowledge compilation converts a formula into a normal form on which queries are tractable that are hard on CNFs [14]. Specifically, we are interested in propositional formulas in Decision-DNNF (or FBDD). Such a formula has following properties:

- *NNF*: A propositional formula is in *negation normal form*, NNF for short, if negations ($\neg$) occur only directly in front of variables and the only other operators are conjunction ($\wedge$) and disjunction ($\vee$) [44].
- *Decomposability*: For any two distinct subformulas C_i, C_j in a conjunction $C = C_1 \wedge \cdots \wedge C_n$ with $i \neq j$, we have $\mathrm{vars}(C_i) \cap \mathrm{vars}(C_j) = \emptyset$, where $\mathrm{vars}(C_i)$ denotes the set of variables that occur in subformula C_i [14].
- *Decision*: Disjunctions are of the form $C = (x \wedge C_1) \vee (\neg x \wedge C_2)$, where x is called *decision variable*. Decision is a stronger version of *determinism*, which requires that subformulas in the disjunction must be logically contradictory [14]. Note that x does not occur in C_1 and C_2 because of decomposability. C_1 and C_2 may be conjunctions.
- *Smoothness*: A disjunction $C = C_1 \vee C_2$ is *smooth*, if $\mathrm{vars}(C_1) = \mathrm{vars}(C_2)$. A formula in C in Decision-DNNF is smooth if all disjunctions in C are smooth. Smoothness is required for model counting [11].

Graph Representation. An NNF formula can be represented as a *directed acyclic graph*, *DAG* for short, where nodes represent $\wedge$, $\vee$, $\top$ (verum) or $\bot$ (falsum) [11, 38]. $\wedge$-nodes are labeled with a set of literals $\mathrm{lits}(N) = \{l_1, \ldots, l_n\}$. For a $\vee$-node $N = N_1 \vee N_2$, there is a decision variable x, such that $x \in \mathrm{lits}(N_1)$ and $\neg x \in \mathrm{lits}(N_2)$. We let the decision literals of N_1 be $\mathrm{declit}(N_1) := x$ and of N_2 be $\mathrm{declit}(N_2) := \neg x$ [14]. For node N, we let $\mathrm{vars}(N)$ be the set of variables that occur in the subgraph rooted at N.

Naturally, a conjunction $C = C_1 \wedge \cdots \wedge C_n$ in an NNF formula corresponds to an $\wedge$-node N in the DAG representation. Node N is labeled with literals occurring in C and has a child node corresponding to each non-literal conjunct. If C is a conjunction of only literals then N has a $\top$-node as its child. A disjunction $C = C_1 \vee C_2$ corresponds to an $\vee$-node

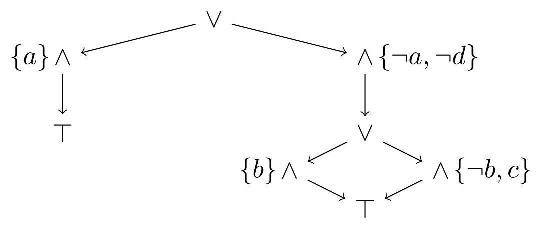


Figure 4 Graph representation of a Decision-DNNF that is equivalent to $F = \{\{a, b, c\}, \{a, \neg d\}\}$. $\wedge$-nodes are labeled with sets of literals.

$N = N_1 \vee N_2$ with child nodes N_1 and N_2 for subformulas C_1 and C_2. An unsatisfiable disjunct corresponds to an $\wedge$-node labeled with the decision literal and that has child node $\bot$. Figure 4 gives an example of a formula in DAG representation. In the following, we use DAG and Decision-DNNF interchangeably.

From Decision-DNNFs to MICE Traces. Assume that the formula is in smooth Decision-DNNF, which can be obtained in polynomial time from a Decision-DNNF [14, Lem A.2]. Then, counting works by evaluating each node in post-order: A $\top$-node has a model count of 1 and a $\bot$-node a count of 0. The model count of an $\wedge$-node is the product of the counts of its children. The model count of an $\vee$-node is the sum of the counts of its children. The count of the root node is the model count of the input formula. To generate a counting trace, we construct a mapping of Decision-DNNF nodes to components and generate claims such that their correctness can be inferred from claims of child nodes.

Annotating and Smoothing the Decision-DNNF

Intuitively, nodes in a Decision-DNNF represent components. While the Decision-DNNF is logically equivalent to the input formula F in CNF, nodes of the Decision-DNNF and clauses in the CNF might not directly be in relation.

Clause Annotation. In DNNFs we can check in polynomial time whether a clause C is entailed. To annotate the NNF, we start at the root node and proceed in preorder. We maintain a set R of clauses, initially set to F, indicating which clauses are remaining. If the current node N is an $\vee$-node, we compute the set of entailed clauses $E = \{C \mid C \in R, N \text{ entails } C\}$ and annotate node N by E. Then, we apply the annotation to the child nodes of N with $R := E$. For $\wedge$-nodes, we construct $R' = R \setminus \{C \mid C \in R, \hat{L} \text{ satisfies } C\}$ by removing the clauses that are solved by $L := \text{lits}(N)$ from the set R. Then, we annotate N by R' and proceed to the children of N with set R'. If we encounter a child node that was already annotated by a different set of clauses, we copy the sub-tree under the child node and proceed with annotating the copy. For $\top$ and $\bot$ nodes, we set $\emptyset$. We denote the annotated clauses for a node N as clauses(N).

Note that this annotation is not equal to the set of entailed clauses of a node. For instance, if an $\vee$-node $N = N_1 \vee N_2$ entails some clauses E_N, the child nodes N_1 and N_2 also entail E_N. But $\wedge$-nodes N_1 and N_2 may be annotated differently because clauses solved by their literals are removed from their annotations.

In the following, we require *non-interfering* annotations for $\wedge$-nodes. We say that annotations of two child nodes *interfere* if one child is annotated by a clause that shares variables with its sibling. The knowledge compilers D4 [38], DSHARP [40], and C2D [12]

Algorithm 2 Decision-DNNF smoothing for Trace Generation

```
Input: An annotated Decision-DNNF with root $N$ and a set $M$ of "missing" nodes.
1  Procedure SMOOTH($N$, $M$)
2      if $N$ is a $\wedge$-node then
3          $P := \Big\{ N' \mapsto \{v \mid v \in M \cap \text{vars}(\text{clauses}(N'))\} \Big| N' \in \text{children}(N) \Big\}$
4          $I := M \setminus \left( \bigcup_{N' \in \text{children}(N)} P(N') \right)$
5          if $\bot \in \mathit{children}(N)$ then
6              return                                     ▷ We do not smooth unsat. nodes.
7          foreach $N' \in \mathit{children}(N)$ do
8              SMOOTH($N'$, $P(N')$)
9          foreach $v \in I \setminus \text{vars}(N)$ do
10             add $(v \wedge \top) \vee (\neg v \wedge \top)$ as child of $N$
11     else if $N = N_1 \vee N_2$ is an $\vee$-node then
12         SMOOTH($N_1$, $M \cup (\text{vars}(N_2) \setminus \text{vars}(N_1))$)
13         SMOOTH($N_2$, $M \cup (\text{vars}(N_1) \setminus \text{vars}(N_2))$)
14     return                                      ▷ Nothing to do for $\top$- and $\bot$-nodes.
```

produce such formulas by construction, as they decompose the input clauses into disjoint components under some partial assignment (dynamic decomposition). Hence, we assume non-interfering annotations for $\wedge$-nodes in the following. Note that we can easily construct non-interfering annotations as follows: If either subformula is unsatisfiable, which is easy to check in Decision-DNNF, it can be replaced by a $\bot$-node, which is not annotated. Otherwise, we apply a transformation from interfering $\wedge$-nodes.

Smoothing. After annotation, we transform the Decision-DNNF into a smooth Decision-DNNF. Therefore, we add additional nodes, such that the children of $\vee$-nodes cover the same variables. While an $\wedge$-node N with variable $v \notin \text{vars}(N)$ can be transformed such that v is contained by replacing N with node $((v \wedge N) \vee (\neg v \wedge N))$ [14], we need a slightly more sophisticated procedure to avoid "invalidating" the annotation. To this end, we preserve the annotation and use one decision node per input variable as done in Algorithm 2. The algorithm takes a node N and a set M of missing variables as input. If N is an $\vee$-node, we run smoothing for both decision branch nodes where set M consists of variables that occur exclusively in the sibling node. If N is an $\wedge$-node, we map children N' of N to sets of variables, where $P(N')$ is the subset of missing variables that occur in some clause in clauses(N'). For two distinct children N_i and N_j of N, $P(N_i)$ and $P(N_j)$ are disjoint since they have non-interfering annotations. If no child is a $\bot$-node, we smooth each child N' with $M := P(N')$. Finally, there remains a subset I of missing variables that do not occur in clauses(N') for any child N'. For each $v \in I \setminus \text{vars}(N)$, we add the node $N_v = (v \wedge \top) \vee (\neg v \wedge \top)$ as a child of N. We label N_v with $\emptyset$. We do nothing for $\top$- and $\bot$-nodes. Our approach ensures that claims generated from child nodes of an $\wedge$-node can be joined (Lemma 22).

Generating Counting Traces

After annotation and smoothing, we have a smooth Decision-DNNF where each node is annotated with a set of clauses. Intuitively, we translate an $\wedge$-node to a join, its literals to an extension, and an $\vee$-node to a composition. To generate a counting trace, we apply the following procedure recursively starting with the root node, which we assume to be a decision.

Tracing $\wedge$-nodes. If N is an $\wedge$-node with parent node N_P, we let $\mathcal{C}_P := (\text{clauses}(N_P), \text{vars}(N_P))$ be the parent component. We consider three cases. In each case, we construct a claim $\mathcal{I}_P$ for the parent component $\mathcal{C}_P$ with assumption lits(N).

1. If N has a $\top$-node as child, we output claim $\mathcal{I}_P = (\mathcal{C}_P, \text{lits}(N), 1)$.
2. If N has a $\bot$-node as a child, we output a claim $\mathcal{I} = (\mathcal{C}_\bot, \emptyset, 0)$ for component $\mathcal{C}_\bot = (F, \emptyset)$. The absence of models statement for $\mathcal{I}$ is trivial since a component with $F \neq \emptyset$ and $V = \emptyset$ is always unsatisfiable. Additionally, we output claim $\mathcal{I}_P = (\mathcal{C}_P, \text{lits}(N), 0)$.
3. Otherwise, N has non-trivial child nodes. We define a join component $\mathcal{C}_J = (\text{clauses}(N), \text{vars}(N) \setminus \text{vars}(\text{lits}(N)))$. If the parent node is not a decision node, we may have lits$(N) = \emptyset$ and $\mathcal{C}_J = \mathcal{C}_P$. First, we output traces for the children of N. From that, we obtain claim $\mathcal{I}' = (\mathcal{C}_{N'}, \emptyset, c')$ for each child N' of N with child component $\mathcal{C}_{N'} = (\text{clauses}(N'), \text{vars}(N'))$. Because the Decision-DNNF is deterministic and we have non-interfering annotations, variables and clauses of child components for distinct children are mutually disjoint. As a result of our smoothing algorithm, no variable in vars(clauses(N')) occurs in another child of N. Hence, we can infer a joint claim $\mathcal{I}_J = (\mathcal{C}_J, \emptyset, c)$ by Lemma 22 (Join), where c is the product of the counts in claims $\mathcal{I}'$. If lits$(N) \neq \emptyset$, we emit a claim $\mathcal{I}_P = (\mathcal{C}_P, \text{lits}(N), c)$ that extends $\mathcal{I}_J$ to the parent component $\mathcal{C}_P$ by Lemma 24.

Finally, if $|\text{lits}(N)| > 1$, we additionally output claim $(\mathcal{C}_P, \{\text{declit}(N)\}, c)$, which can be inferred from $\mathcal{I}_P$ by Lemma 20 (Composition). In our prototype, we use miniSAT to generate a refutation for an absence of models statement supporting this composition. A derivation of lits(N) from declit(N) might be obtained directly in a knowledge compiler.

Tracing $\vee$-nodes. For a decision node $N = N_1 \vee N_2$, consider the component $\mathcal{C}_N = (\text{clauses}(N), \text{vars}(N))$. Again, we first output counting traces for both children N_1 and N_2. Since N_1 and N_2 are $\wedge$-nodes, we obtain claims $\mathcal{I}_1$ and $\mathcal{I}_2$ for $\mathcal{C}_N$ with assumptions $\{\text{declit}(N_1)\}$ and $\{\text{declit}(N_2)\}$. Since declit$(N_1) = \neg\,\text{declit}(N_2)$, we can infer a composed claim $(\mathcal{C}_N, \emptyset, c)$ where c is the sum of the counts of $\mathcal{I}_1$ and $\mathcal{I}_2$. The supporting absence of models statement is straightforward.

Transforming a Decision-DNNF with Interfering Annotations

Current knowledge compilers do not generate Decision-DNNFs where clause annotations of children of $\wedge$-nodes share variables. Nevertheless, we can transform a Decision-DNNF with interfering annotations into one without interfering annotations.

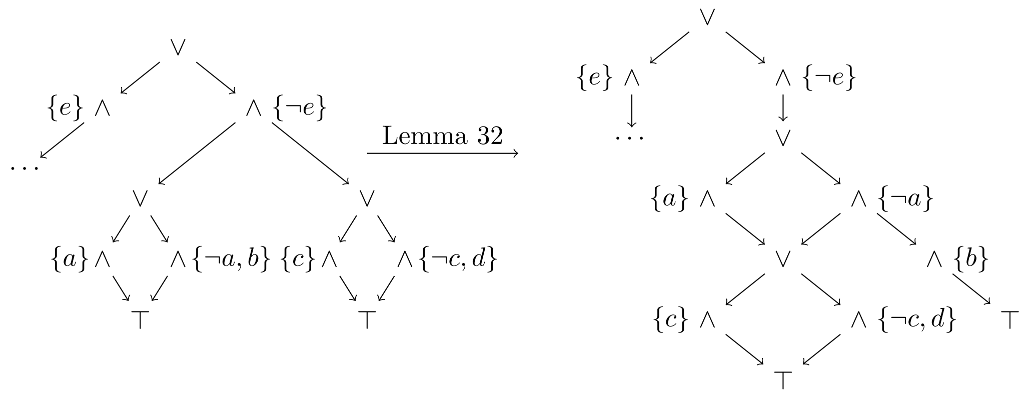
▶ **Lemma 32.** *Let N be an $\wedge$-node with two satisfiable children N_1 and N_2, where N_1 entails a clause C that shares variables $I = \text{vars}(C) \cap \text{vars}(N_2)$ with N_2. Then, we can transform N into a node Decision-DNNF N' with non-interfering annotations. The sub-tree rooted at N' has at most $|I| \cdot (2n+1)$ nodes, where n is the number of nodes in the subtree rooted at N.*

Proof. Consider a literal $\ell \in C|_I$. We construct decision node $N' = N_\ell \vee N_{\neg\ell}$ from N, where $N_\ell = N_1[\ell] \wedge N_2$ and $N_{\neg\ell} = N_1[\neg\ell] \wedge N_2$ with lits$(N_\ell) = \{\ell\}$ and lits$(N_{\neg\ell}) = \{\neg\ell\}$. For a node N_i, $N_i[\ell]$ denotes conditioning of N_i with literal ℓ [14]. If either $N_1[\ell]$ or $N_1[\neg\ell]$ become $\top$ by conditioning, we omit $\top$ and have $N_\ell = N_2$ or $N_{\neg\ell} = N_2$, respectively. We omit N_2 if either $N_1[\ell]$ or $N_1[\neg\ell]$ becomes $\bot$ by conditioning.

As a result, $N_1[\ell]$ is not annotated with the interfering clause C, because C is solved by ℓ in lits(N_ℓ). Further, $N_1[\neg\ell]$ does not contain the interfering variable by construction. If the annotations of $N_1[\neg\ell]$ and N_2 still share variables, we replace $N_1[\neg\ell]$ by $N_1[\neg\ell]'$ obtained in the same way as N'. We proceed until no interfering variable is left, where the node conditioned with $\neg\ell$ becomes $\bot$.

It is easy to see that the result is still a Decision-DNNF. For the number of added nodes, consider that we replace a node N_1 by $N_1[l]$ and $N_1[\neg\ell]$each with at most as many nodes n as the subtree rooted at N. Hence, the replacement for each $\ell \in C|_I$ has at most $2n$ additional nodes, plus one decision node. For each replacement, only one decision branch must be processed further. Thus, N' is replaced by at most $|I| \cdot (2n+1)$ nodes. ◀

▶ **Example 33.** Consider formula $F = \{\{e, a, b\}, \{a, b, c, d\}, \{c, d, f\}, \{\neg f\}\}$, which has an equivalent Decision-DNNF with interfering annotations (left). The first child of the $\wedge$-node labeled with $\{\neg e\}$ is annotated with the first and second clause. The second child is annotated with the second and third clause. Hence, the second clause is an interfering clause. By applying the transformation in Lemma 32 with literal a of the second clause, we obtain a Decision-DNNF (right) that does not lead to interfering annotations.



D A Plain-Text Trace Format

In this section, we describe a trace format that can be efficiently generated by solvers and is simple to implement and process. The format is line-based, where each line starts with a string indicating its type and ends with `0`, followed by the newline character. Within lines, only spaces are allowed as whitespace. Lines starting with `c` are ignored.

```
PREFIX space separated items 0
c ignored line
```

As in the DIMACS format for formulas in CNF, variables are written as positive integers `n` and literals as signed integers `-n` or `n`. A trace starts with a header line, indicating the number of variables `Nv` and clauses `Nc` of the original DIMACS CNF instance.

```
p st Nv Nc 0
```

Clause Definitions

To refer to clauses of the input formula in the trace, each clause is given a unique index. This positive integer index must be unique to the clause. A clause definition is written as a line starting with `f`, followed by the index `idx` and the literals `l1`, …, `ln` of the clause.

```
f idx l1 l2 ... ln 0
```

Component Definitions

Let $\mathcal{C} = (F, V)$ be a component. As with clauses, we assign each component a unique positive integer identifier `id`. This allows referring to the component from other statements in the trace. A component definition line starts with tag `d`, followed by the component identifier `id`, the component's variables $V =$ `v1`$, \ldots,$ `vn`, separator symbol `0`, and the clause indices `c1`$, \ldots,$ `cn` of clauses F.

```
d id v1 v2 ... vn 0 c1 c2 ... cn 0
```

Model Claims

An (exactly one) model claim $(\mathcal{C}, A, 1)$ for component $\mathcal{C} = (F, V)$ is a claim that can be inferred by Lemma 19. It is written as a line starting with tag `m`, followed by the component identifier `id` for $\mathcal{C}$ and the literals `l1`$, \ldots,$ `ln` of its assumption A.

```
m id 1 l1 l2 ... ln 0
```

Join Claims

A join claim $\mathcal{I} = (\mathcal{C}, A, c)$ for component $\mathcal{C} = (F, V)$ is a claim that can be inferred by Lemma 22 or a generalization to n-ary joins. First, we define the subcomponents of $\mathcal{C}$. Note that these definitions apply to all join claims of $\mathcal{C}$, not just a $\mathcal{I}$. A join subcomponent definition starts with tag `jc` followed by the identifier `childn` of a subcomponent of $\mathcal{C}$ and the identifier `id` of $\mathcal{C}$.

```
jc child1 id 0
jc child2 id 0
jc child3 id 0
```

Then, we write a claim line starting with tag `j`, followed by component identifier `id`, the claimed count `c`, and the literals `l1`$, \ldots,$ `ln` of the assumption A.

```
j id c l1 l2 ... ln 0
```

If one claim $\mathcal{I}$ of a joined subcomponent has model count 0, the claims for sibling components of $\mathcal{I}$ may be omitted.

Extension Claims

An extension claim $\mathcal{I} = (\mathcal{C}, A, c)$ for component $\mathcal{C} = (F, V)$ is a claim that can be inferred by Lemma 24 from a claim of subcomponent $\mathcal{C}'$ of $\mathcal{C}$. The claim is written as a line starting with tag `e`, followed by the identifier `id` of $\mathcal{C}$, the identifier `subid` of $\mathcal{C}'$, the claimed count `c`, and the literals `l1`$, \ldots,$ `ln` of the assumption A.

```
e id subid c l1 l2 ... ln 0
```

Absence of Models Statements

An absence of models statement $\mathcal{A} = (\mathcal{C}, A, U, \Delta)$ for a component $\mathcal{C}$ is written as multiple lines. First, the refutation Δ is declared with a unique positive integer identifier `aid`. The declaration line starts with tag `xp` which is followed by the refutation identifier `aid`.

```
xp aid 0
```

Then, the steps δ of Δ are given with one step $\delta = \{l_1, l_2, \ldots, l_n\}$ per line. Each line starts with tag `xs`, followed by the refutation identifier `aid` and the literals `l1`, $\ldots$, `ln` of the refutation step δ.

```
xs idx l1 l2 ... ln 0
```

Finally, we write a statement line finishes. All claims in the set $\mathcal{A}$ states composability for must occur before the statement line. There may be multiple statement lines referencing the same refutation. The statement line starts with tag `xf`, followed by identifier `id` of $\mathcal{C}$, refutation identifier `aid` of Δ, the variables `v1`, $\ldots$, `vn` U, the separator symbol `0`, and the literals `l1`, $\ldots$, `ln` of the assumption A.

```
xf id aid v1 v2 ... vn 0 l1 l2 ... ln 0
```

Composition Claims

A composition claim $(\mathcal{C}, A, c)$ for component $\mathcal{C} = (F, V)$ is a claim that can be inferred by Lemma 20. It is written as a line starting with tag `a`, followed by identifier `id` of $\mathcal{C}$, the refutation identifier of an absence of models statement `aid`, the claimed count `c` and the literals `l1`, $\ldots$, `ln` of the assumption A.

```
a id aid c l1 l2 ... ln 0
```