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. We propose a novel system for certifying model counts. We show how proof traces can be generated for exact model counters based on dynamic programming, counting CDCL with component caching, and knowledge compilation to Decision-DNNF, which are the predominant techniques in today’s exact implementations. We provide proof-of-concepts for emitting proofs and a parallel trace checker.

Based on this, we show the feasibility of using certified model counting in an empirical experiment.

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 [59, 52, 1]. 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, 48, 43, 44, 55, 15, 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 [11, 3]. Model counters are becoming a standard tool for
answering quantitative queries on theories [22], in domains like configuration analysis [57], probabilistic reasoning [1, 45], explainable artificial intelligence [56, 2], risk analysis [62, 17], answer set programming [47, 20, 40], or plausibility [13].

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 [29], 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 [58], which also serves as a basis for several other implementations, like Ganak [55] and Dsharp [46]: If preprocessing is disabled, the solver does not propagate unit clauses present in the input 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 \( F = (a) \land (b) \). 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, 38]. 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 [38]. 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 [32], 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 [33]. 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 [30, 38, 61, 10], among them the popular format DRAT [61]. 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 [44] and Dsharp [46], 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 [37], 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 [42]. 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 [12], 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 [50, 31] instead.

2 Preliminaries

We assume that the reader is familiar with basic notions on functions, set theory, computational complexity [49], and propositional logic [41].

Propositional Satisfiability and Model Counting (\#SAT). A literal $\ell$ is a propositional variable $v$ or its negation $\neg v$. Conversely, we refer to the variable $v$ of literals $v$ and $\neg v$ by $\text{var}(v) := v$ and $\text{var}(\neg v) := v$, 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 $\text{vars}(C)$ to the variables occurring in a finite set $C$ of literals, i.e., $\text{vars}(C) := \{\text{var}(\ell) \mid \ell \in C\}$. 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 $\text{vars}(F)$ denote the variables occurring in a formula, i.e., $\text{vars}(F) := \bigcup_{C \in F} \text{vars}(C)$. 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 $V$.
30:4 Proofs for Propositional Model Counting

We used the negation of a found model to the formula, thereby forbidding it. After solving outputs that
an outputted model count. To this end, we systematically enumerate models and add the
more general redundancy property based on extended resolution instead of entailment [35].

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

Certified SAT Solving. Certified results are common in SAT solving [30, 35, 28] and support
of a standardized format is mandatory for solvers taking part in the competition [28]. Popular
formalisms like DRAT [35] and RUP are examples of the more general notion of propositional
proof systems [9]. Let \( \Sigma \) be an alphabet, \( L \subseteq \Sigma^* \) be language, and 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 \( s : \Sigma_1^* \rightarrow \Sigma^* \) with range \( L \) that maps
from words over a proof alphabet \( \Sigma_1 \) 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 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 TAUT for a
given formula. Since SAT solving usually works on CNF formulas as input, practical focus is
on clause proofs. Clausal proofs are sequences of clauses, where each clause is entailed by
the original formula [32]. 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) [30] 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 [35].

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, 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 $F = (a \lor b) \land (c \lor d)$, which has 9 models. In detail, there are 3 assignments to $a$ and $b$ that satisfy $(a \lor b)$ and 3 assignments to $c$ and $d$ to satisfy $(c \lor d)$. 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 [33]. 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., $\text{Mod}(F) = \text{Mod}(a \lor b) \times \text{Mod}(c \lor d)$. Hence, the model count of formula $F$ can simply be expressed as the product of the cardinality of both sets, i.e., $|\text{Mod}(F)| = |\text{Mod}(a \lor b)| \cdot |\text{Mod}(c \lor d)|$.

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.
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 $C = \{\{a,b,c\}, \{a,\neg d\}\} \cup \{\{a,b,c,d\}\}$ be a component, $(C, \{a,b\}, 4)$ and $(C, \{a,d\}, 4)$ be claims of $C$. 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 $C = (F, V)$ be a component and $S$ be a set of claims over $C$.
  - Then, $S$ is non-overlapping if, for every two distinct claims $(C, A_1, c_1)$ and $(C, A_2, c_2)$ in set $S$, we have $\text{Mod}_{A_1}(C) \cap \text{Mod}_{A_2}(C) = \emptyset$.
  - Let $U \subseteq V$. If $\text{vars}(A') = U$ for every $(C, A', c') \in S$, we call $S$ uniform for $U$.

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

- **Observation 10 (⋆).** Let $C = (F, V)$ be a component, $U \subseteq V$ a set of variables, and $S$ a set of claims over $C$. 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 $C = \{\{a,b,c\}, \{a,\neg d\}\} \cup \{\{a,b,c,d\}\}$ and claims $I_1 = (C, \{a,d\}, 4)$, $I_2 = (C, \{a,\neg d\}, 4)$ of $C$. $I_1$ and $I_2$ are non-overlapping, but cannot correctly be combined to $I = (C, \emptyset, 8)$, because models with $a \mapsto 0$ are counted by neither $I_1$ nor $I_2$. However, adding a third claim $(C, \{\neg a, \neg d\}, 3)$ is sufficient to cover $C$ 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 $C = (F, V)$ be a component and $S$ a set of claims over $C$. For an assumption $A$, we call $S$ exhaustive for $A$ if for every model $\alpha \in \text{Mod}_{A}(C)$ there is a claim $(C, A', c') \in S$ with $\alpha$ satisfies $A'$ and $A \subseteq A'$.

Unfortunately, already the task of checking exhaustiveness is hard.

- **Proposition 13 (**,Exhaustiveness of Claims is co-NP Hard).** Let $S$ be a set of claims for a component $C = (F, V)$ that is uniform for $U \subseteq V$ and $A$ be an assumption with $\text{vars}(A) \subseteq U$. 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 $\#\text{SAT}$ is known to be in $\#P$ [59], 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.

---

1 Proofs of statements marked with ⋆ can be found in Appendix A.
Definition 14 (Absence of Models Statement). Let $C = (F, V)$ 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 $(C, A', c) \in S$. Then, given a clausal proof $\Delta$, we call $A = (C, A, U, \Delta)$ an absence of models statement. Such a statement $A$ is correct if $\Delta$ is a refutation of $\hat{A} \cup \{C|_V \mid C \in F\} \cup \{\neg A' \mid (C, A', c') \in S\}$.

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.

Lemma 15 (*, Absence of Models). Let $A = (C, A, U, \Delta)$ be an absence of models statement and $S$ be a set of claims for component $C$ that is uniform for $U$ and we have $A \subseteq A'$ for every claim $(C, A', c) \in S$. If $A$ 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 $C = (\{\{a, b, c\}, \{a, \neg d\}\}, \{a, b, c, d\})$ and a set $S$ of claims of $C$ 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 $C$ has no model with $a \rightarrow 0$ and $d \rightarrow 1$, $S$ is exhaustive for $\emptyset$. To prove this, we can easily find a refutation $\Delta$ for $\{\ldots, \{a, \neg d\}\} \cup \{\neg a, \neg d\}, \{\neg a, d\}, \{a, d\}$). Hence, we can construct a correct absence of models statement $(C, \emptyset, U, \Delta)$ 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 $C = (F, V)$, that is composable to $A$ and uniform for $U$. Let $A_s$ be an assumption with $A \subseteq A_s$ and $\text{vars}(A_s) \subseteq U$. Then, $S_{A_s} := \{(C, A', c') \in S \mid A_s \subseteq A'\}$ 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 $T = \langle s_1, \ldots, s_n \rangle$ 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 $I = (C, \emptyset, c)$ in $T$ for a component $C = (F, \text{vars}(F))$. 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 succinctly expressed by claims. We denote a rule as premises and inference separated by a double rule. Intuitively, these rules establish the correctness of a claim by combining claims of sub-components or claims with stronger assumptions. 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 model, composition, join, and extension. Composition itself is backed up by exhaustiveness, thereby using the sufficient absence of models statement via clausal proofs.
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.

\[ \tau_A \text{ satisfies } F \]

\[ ((F, \text{dom}(\tau_A)), A, 1) \]

\[ \text{Lemma 19 (⋆, Exactly One Model). Let } C = (F, V) \text{ be a component and } (C, A, 1) \text{ be a claim for } C \text{ with } \text{vars}(A) = V. \text{ The claim is correct if and only if } \tau_A \text{ satisfies } F. \]

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 $A$. The absence of models statement contains a resolution refutation (see Def 14). 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 $(C, A_i, c_i)$ in $S$ as a requirement.

Claims $S = \{(C, A_1, c_1), \ldots, (C, A_n, c_n)\}$ are correct, $U = \text{vars}(A_1) = \cdots = \text{vars}(A_n)$

\[ (C, A, \Sigma_{1 \leq i \leq n} c_i) \]

\[ \text{Absence of models statement } (C, A, U, \Delta) \text{ is correct for } S \]

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

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.

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

\[ \text{set } S:\]

\begin{center}
\begin{tabular}{c|c}
\hline
4 & $\{a \quad d\}$ \\
4 & $\{a \quad \neg d\}$ \\
3 & $\neg a \quad \neg d$ \\
\hline
\end{tabular}
\end{center}

\[ \text{composed claim:} \]

\[ \begin{tabular}{c|c}
\hline
11 & $\emptyset$ \\
\hline
\end{tabular} \]

In addition, we know that $\emptyset$ is composable to $A = \{\neg a, d\}$. By composition, we infer a claim $(C, A, 0)$. This is equivalent to stating that $C$ is unsatisfiable under assumption $A$. 
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 \(((F_2, V_2), A_2, c_2)\).

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

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 \(C = (F, V)\) be a component; \(I = (C, A, c)\) be a claim; \(C_1 = (F_1, V_1)\) and \(C_2 = (F_2, V_2)\) be sub-components of \(C\) 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 \(C \in F_1\) has \(\text{vars}(C) \cap (V \setminus V_1) = \emptyset\). If \(I_1 = (C_1, A|_{V_1}, c_1)\) and \(I_2 = (C_2, A|_{V_2}, c_2)\) are correct claims over \(C_1\) and \(C_2\) and \(c = c_1 \cdot c_2\), then \(I\) is correct, i.e., \(|\text{Mod}_A(C)| = c\).

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 \(C_1 = (F_1, V_1)\) and \(C_2 = (F_2, V_2)\) with \(F_1 = \{a, b, c\}\), \(V_1 = \{a, b, c\}\), \(F_2 = \{a, d\}\), and \(V_2 = \{a, d\}\). The tables below represent claims for both components, along with claims for a third component \(C = (F_1 \cup F_2, V_1 \cup V_2)\). All claims of \(C\) are inferred from claims of sub-components \(C_1\) and \(C_2\) using Lemma 22.

<table>
<thead>
<tr>
<th>claims of (C_1):</th>
<th>claims of (C):</th>
<th>claims of (C_2):</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>(A)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{a, b}</td>
<td>\cdots</td>
</tr>
<tr>
<td>2</td>
<td>{a, \neg b}</td>
<td>\cdots</td>
</tr>
<tr>
<td>2</td>
<td>{\neg a}</td>
<td>\cdots</td>
</tr>
<tr>
<td>1</td>
<td>{\neg a, \neg b}</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

**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 \(((F', V'), A', c)\).

\[
((F', V'), A', c) \text{ is correct}
\]
\[
\tau_A \text{ satisfies } F \setminus F', \text{ where } F \supseteq F' \text{ and } A|_{V'} = A' \text{ and } V' \subseteq V
\]
\[
\forall C \in F' : \tau_A|_{V \setminus V'} \text{ does not satisfy } C
\]
\[
((F, V), A, c)
\]

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 \(C = (F, V)\) be a component, \(I = (C, A, c)\) be a claim, \(C' = (F', V')\) be a sub-component of \(C\), and \(I' = (C', A|_{V'}, c)\) be a correct claim. If \(V \setminus V' \subseteq \text{vars}(A)\), \(\tau_A \text{ satisfies } F \setminus F'\), and \(\tau_A|_{V \setminus V'} \text{ does not satisfy } C\) for every clause \(C \in F'\), then \(I\) 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 \(I'\) requires a corresponding model in the extended claim \(I\). We ensure this by enforcing that clauses that are only in the larger component \(C\), are satisfied by the literals added to the assumption. Conversely, every model of the claim \(I\), when restricted to the variables of \(C'\), must be a model of \(I'\). Intuitively, this establishes a sufficient one-to-one correspondence between the models of \(I\) and \(I'\).

**Example 25.** We extend a component \(C' = (F', V')\) with \(F' = \{a, b, c\}\) and \(V' = \{a, b\}\) to the component \(C = (F, V)\) 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 \(C\) can be verified by Lemma 24 (extension). Note that the extended assumption must include \(\neg c\) to satisfy extension.

<table>
<thead>
<tr>
<th>(c)</th>
<th>(A)</th>
<th>(c)</th>
<th>(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a, b}</td>
<td>1</td>
<td>{a, b, \neg c, d}</td>
</tr>
<tr>
<td>1</td>
<td>{a, \neg b}</td>
<td>1</td>
<td>{a, b, \neg c, \neg d}</td>
</tr>
<tr>
<td>1</td>
<td>{\neg a, b}</td>
<td>1</td>
<td>{a, \neg b, \neg c, d}</td>
</tr>
<tr>
<td>1</td>
<td>{\neg c, \neg d}</td>
<td>1</td>
<td>{a, \neg b, \neg c, \neg d}</td>
</tr>
</tbody>
</table>

**Remark 26.** Let \(C = (\emptyset, \emptyset)\) be a component. Since \(\text{Mod}_0(C) = \{\emptyset\}\), claim \(I = (C, \emptyset, 1)\) is correct. Thus, a claim satisfying exactly one model can be expressed as an extension of \(I\).

### 3.4 Proofs for Model Counting

Above, 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.

**Definition 27 (MICE Proof Step).** Let \(T = (s_1, \ldots, s_n)\) be a model counting trace for a given formula \(F\) and \(S = \{s_1, \ldots, s_n\}\). We say \(s_1 \not\in S\) 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 \((C, A, U, \Delta)\) that is correct for \(\{(C, A', c') | (C, A', c') \in S, A \subseteq A', \text{vars}(A') = U\}\) (cf., Lemma 15);
- \(s_r\) is a claim of exactly one model; (cf., Lemma 19);
- \(s_r\) is a claim joining two claims \(I_1, I_2 \in S\) (cf., Lemma 22);
- \(s_r\) is a claim extending another claim \(I \in S\) (cf., Lemma 24); or
- \(s_r\) is a claim \((C, A, c)\) by composing a set \(S' \subseteq S\) of claims and there is an absence of model statement \((C, A, U, \Delta)\) 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 \(T = (s_1, \ldots, s_n)\) 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 lose completeness, i.e., a MICE proof exists for any formula.
Algorithm 1 Simple Trace Correctness Checking.

Input: A model counting trace $T = \langle s_1, \ldots, s_n \rangle$

Output: “Correct” if $T$ is a MICE proof, “Incorrect” otherwise

1. for $i \in \{1, \ldots, n\}$ do

   2. $P := \{s_1, \ldots, s_{i-1}\}$ // Set of predecessors.

   3. if $s_i$ is an absence of models statement $(C, A, U, \Delta)$ then

      4. $S := \{(C, A_j, c) \in S \mid A_j \supseteq A, \text{vars}(A_j) = U\}$

      5. $F_E := \hat{A} \cup \{C \mid V \mid C \in F\} \cup \{\neg A' \mid (C, A', c') \in S\}$

      6. if $\Delta$ is refutation of $F_E$ then continue

   else if $s_i$ is a claim $(C, A, c)$ with $C = (F, V)$ then

      7. if $\text{vars}(A) = V$ and $c = 1$ and $\tau_A$ satisfies $F$ then continue” // Lemma 19

      8. else if $s_i$ is correct by joining $s_j, s_k \in P$ then continue” // Lemma 22

      9. else if $s_i$ is correct by extending $s_j \in P$ then continue” // Lemma 24

   else if $P$ contains some $(C, A, U, \Delta)$ then

      10. $S := \{(C, A_j, c) \in S \mid A_j \supseteq A, \text{vars}(A_j) = U\}$

      11. if $s_i$ is correct by composition of $S$ then continue” // Lemma 20

   return Incorrect” // Step verification failed.

14. return Correct

Theorem 29 (⋆,Soundness & Completeness). Given formula $F$, component $C = (F, \text{vars}(F))$.

- Soundness: If $T$ is a MICE proof containing claim $(C, \emptyset, c)$, then $F$ has $c$ 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.

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 and call it SHARPCHECK, which is open-source and available on github:vroland/sharptrace. For space reasons, we will 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 [54], dynamic programming [53, 24, 34, 16], or knowledge compilation [12, 44]. 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 [58]. It serves as basis for several model counters, e.g., Dsharp [46] or GANAK [55]. 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](https://github.com/vroland/sharpSAT/tree/proof-trace).

**DPDB** The solver implements dynamic programming algorithms on TDs using database management systems [26]. We extended the model counting implementation to translate result tables into claims and use MINISAT [18] to generate refutations. Our implementation is available online at [https://github.com/vroland/dp_on_dbs/tree/sharpsat_proof](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 will provide conceptual details for modifying a knowledge compiler in an extended version. Our implementation is available online at [https://github.com/vroland/nnf2trace](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.

![Figure 2](https://example.com/image.png)  
**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.
Table 1 Performance of solving and trace checking. \# refers to number of instances, \( t \) the sum of instance runtimes, size the total proof size, \( O \) without tracing, \( T \) with tracing, and \( V \) checking.

(a) Number of solved instances.

<table>
<thead>
<tr>
<th>Solver</th>
<th>#( O )</th>
<th>#( T )</th>
<th>#( V )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARPSAT</td>
<td>141</td>
<td>67</td>
<td>66</td>
</tr>
<tr>
<td>DPDB</td>
<td>137</td>
<td>78</td>
<td>65</td>
</tr>
<tr>
<td>D4</td>
<td>172</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Solver</th>
<th>( t_O ) [h]</th>
<th>( t_T ) [h]</th>
<th>( t_V ) [h]</th>
<th>size [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPDB</td>
<td>65:23:27</td>
<td>2:44:45</td>
<td>1:29:28</td>
<td>17</td>
</tr>
<tr>
<td>D4</td>
<td>15:00:13</td>
<td>0:23:28</td>
<td>0:00:46</td>
<td>1</td>
</tr>
</tbody>
</table>

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 [23].

Hardware, Measure, and Restrictions. All solvers run on a server with two physical Intel Xeon Silver 4112 CPUs, 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. We follow guidelines for empirical evaluations [60, 27, 25].

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

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 \( \approx 750 \) 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. For sharpSAT, verifying proofs takes more total runtime than running sharpSAT with tracing.

**Discussion.** When verifying traces from sharpSAT, most time is spent on parsing and validating the integrity of the trace, due to their size. Then, 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 [39, 36].
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 \textsc{#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 \textsc{#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 [14]. Further, establishing more general, but efficiently verifiable, inference rules may facilitate integration into solvers. Here, stronger proof techniques might come in handy [19, 31]. 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


\textbf{Omitted Proofs}

\textbf{(Re-)combining Claims of Search Spaces}

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

\textbf{Proof.} Recall that since $S$ is uniform for $U$, we have $\text{vars}(A) = U$ for all claims $(C, A, c)$ in $S$. Then, for every assignment $\alpha$ that is total over $V$, there is exactly one assumption $A$ with $\text{vars}(A) = U$ and $\alpha$ 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 $\alpha$ to $V$, there is exactly one matching assumption over variables $U$, thus there is at most one claim $(C, A, c)$ in $S$ with $\alpha \in \text{Mod}_A(C)$. \hfill $\blacksquare$
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 \( \text{vars}(A) \subseteq U \). 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 \( \mathcal{C} = (F, \text{vars}(F)) \) and assumption \( \emptyset \). We know that \( F \) is in UNSAT if and only if \( \text{Mod}_{\emptyset}(\mathcal{C}) = \emptyset \). 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 \( \text{vars}(A) \subseteq U \), and \( S \) a set of claims of component \( \mathcal{C} \) with assumptions over variables \( U \). Then, a function \( r \) that outputs the formula \( F_E := \hat{A} \cup \{ C | C \in F \} \cup \{ \neg A' | (C, A', c') \in S \} \) from \( S, \mathcal{C}, \) and \( A \) is a reduction to UNSAT by Lemma 15: From Definition 14 and Lemma 13 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 \( \{ C | C \in F \} \), where we need \( O(|C| \cdot |V|) \) steps per clause \( C \in F \). Finally, we output \( \neg A' \) for every \( (C, A', c') \in S \), 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 for \( U \) and we have \( A \subseteq A' \) for every claim \((C, A', c) \in S\). 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 \( \Delta \) is a refutation of \( E = \hat{A} \cup \{ C | C \in F \} \cup \{ \neg A' | (C, A', c') \in S \} \). For proof of contradiction, assume \( S \) is not exhaustive for \( A \). Then, there is a model \( \alpha \in \text{Mod}_{A}(\mathcal{C}) \), but there is no claim with assumption \( A_{e} \in S \), such that \( \tau_{A_{e}} = \alpha|_{U} \). Since \( \alpha \in \text{Mod}_{A}(\mathcal{C}) \), assignment \( \alpha \) satisfies \( \hat{A} \cup \{ C | C \in F \} \). Hence, assignment \( \alpha \) must falsify \( \{ \neg A' | (C, A', c') \in S \} \), or else \( E \) is satisfiable and \( \Delta \) is not a refutation. However, since no claim with \( A_{e} \) is in \( S \), \( \alpha \) satisfies \( \neg A' \) for all assumptions \( A' \) of claims in \( S \) by construction. Thus, \( \alpha \) satisfies \( E \), which contradicts that \( \Delta \) be a refutation. Hence, if \( \Delta \) 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 \( \text{vars}(A) = V \). The claim is correct if and only if \( \tau_{A} \) satisfies \( F \).

Proof. Consider the assignment \( \alpha = \tau_{A} \). Since \( \text{vars}(A) = V \), assignment \( \alpha \) is total over \( V \) and \( \text{Mod}_{A}(\mathcal{C}) \subseteq \{ \alpha \} \). Consequently, \( \tau_{A} \) satisfies \( F \) if and only if we have \( \text{Mod}_{A}(\mathcal{C}) = \{ \alpha \} \) and \( |	ext{Mod}_{A}(\mathcal{C})| = 1 \).
Lemma 20 (Composition or Unsatisfiability). Let $C = (F, V)$ be a component, $I = (\ell, A, c)$ be a claim with assumption $\ell$, $S$ be a set of claims over $C$ that is composable to $A$, and $c := \sum_{(C, A', c') \in S} c'$. If every claim in $S$ is correct, then $I$ is correct, i.e., $|\text{Mod}_A(C)| = c$.

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

Lemma 22 (Join). Let $C = (F, V)$ be a component; $I = (\ell, A, c)$ be a claim; $C_1 = (F_1, V_1)$ and $C_2 = (F_2, V_2)$ be sub-components of $C$ 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 $C \in F_1$ is in $\text{Mod}_{A|V_1}(C_1)$ and $\text{Mod}_{A|V_2}(C_2)$ similarly as in related work [53]. Since $|J| = |\text{Mod}_{A|V_1}(C_1)| \cdot |\text{Mod}_{A|V_2}(C_2)| = c_1 \cdot c_2$, we then conclude that $|\text{Mod}_A(C)| = c_1 \cdot c_2$.

First, we establish that $f$ is a mapping from $\text{Mod}_{A}(C)$ to $J$. For any $\alpha \in \text{Mod}_{A}(C)$, $\alpha|_{V_1}$ is in $\text{Mod}_{A|V_1}(C_1)$ 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 $\alpha$ that are also in $\alpha|_{V_1}$. By the same reasoning, we have $\alpha|_{V_2} \in \text{Mod}_{A_2}(C_2)$.

Further, $f$ is injective, because for every $\alpha_1, \alpha_j \in \text{Mod}_{A}(C)$ 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 $V_1 \cap V_2 \subseteq \text{vars}(A)$ and $A$ is consistent by definition. Since $\alpha$ satisfies every clause $F_1 \alpha_2$ satisfies every clause $F_2$, and $\alpha$ is consistent, $\alpha$ satisfies every clause in $F_1 \cup F_2 = F$. In consequence, $\alpha \in \text{Mod}_{A}(C)$.

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 $C = (F, V)$ be a component, $I = (\ell, A, c)$ be a claim, $C' = (F', V')$ be a sub-component of $C$, and $I' = (\ell', A|_{V'}, c)$ be a correct claim. If $V \setminus V' \subseteq \text{vars}(A)$, $\tau_A$ satisfies $F \setminus F'$, and $\tau_{A|V'}$ does not satisfy $C$ for all clauses $C \in F'$, then $|\text{Mod}_{A}(C)| = c$.

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

To see that $f$ is a valid mapping, we show that if $\alpha$ is a model of $C$, $\alpha|_{V'}$ 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 $\alpha|_{V'} \in \text{Mod}_{A|V'}(C')$ for all $\alpha \in \text{Mod}_{A}(C)$ and $f$ is a valid mapping.

To show that $f$ is injective, consider $\alpha_1, \alpha_2 \in \text{Mod}_{A}(C)$, 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 $\gamma$ satisfies $\tilde{A}$, we have $\alpha_i = \alpha_i|_{V'} \cup \gamma = \alpha_j|_{V'} \cup \gamma = \alpha_j$. □
For surjectivity, let \( \beta \in \text{Mod}_{\mathcal{A}|_V, V'}(\mathcal{C}') \) be a model of claim \( T' \). Because \( V \setminus V' \subseteq \text{vars}(A) \), we can construct an assignment \( \alpha \) as \( \beta \cup \tau_A \). Then, \( \beta \) satisfies the clauses \( F' \), \( \tau_A \) satisfies the clauses \( F \setminus F' \), thus \( \alpha \) is an assignment over the variables \( V \) that satisfies the clauses \( F \). Hence, we have \( \alpha \in \text{Mod}_A(\mathcal{C}) \).

Since \( f \) is bijective and \( F' = (\mathcal{C}', A|_V, \cdot) \) is a correct counting trace, we have that \( |\text{Mod}_{\mathcal{A}|_V, V'}(\mathcal{C}')| = |\text{Mod}_A(\mathcal{C})| = c \). 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 \( A = (\mathcal{C}, A, U, \Delta) \) in \( S \). Since \( A \) is a MICE step from its predecessors, \( A \) 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 & Completeness).** Given formula \( F \), component \( \mathcal{C} = (F, \text{vars}(F)) \).

- **Soundness:** If \( T \) is a MICE proof containing claim \( (\mathcal{C}, \emptyset, c) \), then \( F \) has \( c \) 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 \( I = (\mathcal{C}, \emptyset, c) \), the claim \( I \) is correct. Hence, \( |\text{Mod}_A(\mathcal{C})| = c = |\text{Mod}(F)| \).

Next, we show completeness: Let \( N := \text{Mod}(F) \) 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 \( S := \{(\mathcal{C}, A_n, 1) \mid \alpha \in N\} \) where \( \tau_{A_n} = \alpha \). 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 \( E = \{\mathcal{C}[V \mid C \in F]\} \cup \{\lnot A' \mid (\mathcal{C}, A', c') \in S\} \). Hence, there is a refutation \( \Delta \) for \( E \) and a correct absence of models step \( A = (\mathcal{C}, A, V, \Delta) \). Such a refutation \( \Delta \) must exist for every unsatisfiable formula \( E \) [51]. Since \( S \) is uniform for variables \( V \), and exhaustive for \( \emptyset \), it is composable to \( \emptyset \). Hence, the claim \( I = (\mathcal{C}, \emptyset, c) \) where \( c = |N| \) can be inferred by Lemma 20. Thus, \( I \) is a MICE step from claims \( S \) and absence of models statement \( A \).

Finally, we construct a trace \( T = \langle I_1, \ldots, I_n, A, I \rangle \) with \( I_1, \ldots, I_n \in S \). It is easy to see that \( T \) is complete for \( F \) since it contains \( I \). Since \( I_1, \ldots, I_n \) are correct by MICE steps from \( \emptyset \) and \( I \) is a MICE step from steps \( \{I_1, \ldots, I_n, A\} \), 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 $(C, A, U, \Delta)$, 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 $F_E := \hat{A} \cup \{C | C \in F\} \cup \{\neg A' | (C, A', c') \in S\}$. 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 $\Delta$ 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 $\text{vars}(\cdot)$ 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 $(C, A, U, \Delta)$ 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.