DivKC: A Divide-and-Conquer Approach to Knowledge Compilation

Olivier Zeyen[1], Karim Tit[1], Maxime Cordy[1], and Gilles Perrouin[2]

[1] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
[2] PReCISE/NaDI, University of Namur, Belgium

Abstract. Knowledge compilation (KC) transforms Boolean formulae into alternative representations that allow for more efficient reasoning. However, KC still fails to scale on some Boolean formulae, including some representing the variability of large configurable systems (e.g., OS kernels, automotive product lines, etc.), for which these analyses are paramount. We hypothesise that a divide-and-conquer strategy to knowledge compilation may push its scalability further. Concretely, our DivKC algorithms decompose a large Boolean formula into two smaller ones, which we can easily compile into the d-DNNF form. When evaluated on a diversified benchmark of 4,656 formulae, DivKC compiles 114 formulae out of the 672 formulae that were previously out of reach for the D4 state-of-the-art d-DNNF compiler. We then show how to leverage DivKC decompositions to build an approximate model counter and a uniform random sampler.

1 Introduction

Knowledge compilation (KC) is the task of transforming Boolean formulae into alternative representations that allow for more efficient reasoning [7]. Boolean formulae are typically expressed in conjunctive normal form (CNF), which facilitates specific operations such as conditioning and conjunction. However, fundamental tasks such as model counting (#SAT) and uniform random sampling (URS) remain computationally intractable for large CNF instances. URS plays an important role in software product line (SPL) analysis, where it is used for tasks such as software testing, statistical inference, and optimal configuration search [18]. Knowledge compilation can help overcome the intractability of URS and #SAT by transforming CNF formulae into representations that are more amenable to #SAT [28] and URS [23].

One such language is the deterministic decomposable negation normal form (d-DNNF) [6], which is known to scale well in practice [27]. Efficient compilers for the d-DNNF language exist — most prominently D4 [14] — thereby enabling efficient solving of #SAT and URS. Despite these significant advances, formulae with large and intricate solution spaces remain out of reach for existing compilers [27]. Approximate algorithms exist for #SAT like ApproxMC 7 [21]. However, approximate model counting does not generate the reusable data

structures that KC offers. Therefore, an approximate algorithm can be unsuitable if multiple calls to a model counter are necessary (e.g., feature cardinality computation [28]) or for problems that necessitate solving other reasoning tasks.

In order to enhance the effectiveness of KC, we propose DivKC, a divide-and-conquer approach for d-DNNF compilation. The key principle of our method is to decompose an input formula F to produce two smaller formulae that can be compiled independently and at a lower computational cost than F. This decomposition brings many advantages, including its application to #SAT and URS and the production (by construction) of sound lower and upper bounds for the model count of F. We combine these advantages into an effective statistical method to estimate $|R_F|$, the number of models of F. This method relies on computing approximate lower and upper bounds for $|R_F|$, which are shown to be tighter than the two bounds obtained during the decomposition. As for URS, we can similarly simplify the resolution of this problem by successfully sampling from the two decomposed formulae.

To assess the benefits of our approach, we conduct extensive experiments on four datasets totalling 4,656 formulae. By using our method, we manage to compile to d-DNNF 114 formulae out of the 672 formulae that were previously out of reach for D4 [14]. We thereby demonstrate that DivKC can enhance the compilation ability of the state-of-the-art d-DNNF compiler. Moreover, we show that our statistical method to compute upper and lower bounds of $|R_F|$ achieves 85% coverage of the true model count, while producing intervals that are significantly smaller than the theoretical bounds (about 9.5 billion times smaller for the median case). Finally, we show that our random sampler based on DivKC is the first heuristic-based random sampler to validate at least one test of the test suite proposed by Zeyen et al. [34]. All of the programs and experimental results are available on our companion GitHub [32].

2 Related Work

Model counting is the problem of counting the number of solutions to a formula. Model counting can be done either by specialised algorithms like sharpSAT [29] or GANAK [22], or by using knowledge compilers. Knowledge compilation can generate a reusable data structure that accelerates computations, as shown by Sundermann et al. [28]. Approximate model counters have been proposed to overcome the lack of scalability of exact model counting. A notable approximate model counter is ApproxMC 7 [21], which provides theoretical guarantees on the quality of the model counts and is generally considered to be the state-of-the-art in approximate model counting. Other approximate model counters are ApproxCount [31], which offers no guarantees, and SampleCount [11], which returns a lower bound to the true model count with high confidence. However, approximate model counting has the disadvantage of not generating a reusable data structure. Therefore, an approximate model counter is unlikely to be suitable if multiple calls to a model counter are necessary.

Uniform random sampling is a problem related to model counting, as model counting can be used to sample uniformly at random from a formula, *e.g.*, SPUR [1] and Smarch [19]. A uniform random sampler is a procedure that, given a formula F, will return a solution to F at random. There exist heuristic-based samplers like STS [9], and Quicksampler [8] or true uniform random samplers like SPUR [1]. A notable uniform random sampler is KUS [23], which uses D4 [14] to compile a formula to d-DNNF and then uses the compiled form to sample solutions of F.

Sundermann et al. [28] showed that knowledge compilation can be instrumental in practice but also exposed its limitations [27]. Many knowledge compilers exist, such as C2D [5], D4 [14], and DSharp [17]. While all of them scale to many large formulae, some remain out of reach for any of these compilers.

DMC [15] is a distributed model counter based on D4 [14]. DMC distributes the workload similarly to work stealing. Worker nodes try to solve the problem and notify the master node when they are idle. If a worker node is idle, the master suggests help to busy worker nodes. A worker node can either accept the help and delegate some work or reject the help.

Given the many uses of knowledge compilation and the limitations of current approaches, we propose an algorithm which allows us to push the limits of current knowledge compilers by splitting the input formula into smaller, more manageable parts before using the state-of-the-art D4 [14] compiler. Importantly, our work differs from DMC because we do not use variable assignments to split the input formula. Consequently, we only use two calls to D4 [14] to obtain our final d-DNNF.

3 Preliminaries

A Boolean formula F is defined over a set of Boolean variables $\mathit{Var}(F)$ and evaluates to either true or false. A literal is either a variable x or its negation $\neg x$. The notation $\mathit{Var}(x)$ (or $\mathit{Var}(\neg x)$) refers to the variable associated with the literal x (or $\neg x$, respectively).

A formula F is in negation normal form (NNF) if the negation only appears in front of variables. A clause is a disjunction of literals and can be represented as a set of literals. F is in conjunctive normal form (CNF) if F is written as a conjunction of clauses ($F = \bigwedge_{C_i} \bigvee_{l \in C_i} l$). A CNF formula can be represented as a set of clauses.

An assignment a to the variables $\mathit{Var}(F)$ is a set of literals such that $\forall x \in a : (\neg x \notin a)$. An assignment a is a partial assignment if $\exists x \in \mathit{Var}(F) : (x \notin a \wedge \neg x \notin a)$. An assignment a is a complete assignment if $\forall x \in \mathit{Var}(F) : (x \in a \vee \neg x \in a)$. A model m of F ($m \models F$) is a complete assignment such that F evaluates to true under m. We define R_F as the set of models m of F such that $m \in R_F$ if and only if $m \models F$. We define $|R_F|$ as the number of models of F. A partial assignment a is sufficient if for a CNF formula F we have $\forall c \in F : (c \cap a \neq \emptyset)$, i.e., any complete assignment b with $a \subseteq b$ is a model of F ($b \models F$). Two assignments a, b are orthogonal if they disagree on at least one literal, i.e., $\exists x \in a : (\neg x \in b)$. A

set of partial assignments is orthogonal if and only if every pair of assignments is orthogonal. We denote by $F|_a$ the conditioning of F with a (i.e., the propagation of the literals of a in F).

F is in deterministic decomposable NNF (d-DNNF) if every conjunction is *decomposable* and every disjunction is *deterministic*. A conjunction $\bigwedge A_i$ is *decomposable* if $\forall i \neq j : (Var(A_i) \cap Var(A_j) = \emptyset)$. A disjunction $\bigvee O_i$ is *deterministic* if $\forall i \neq j : (R_{O_i \wedge O_j} = \emptyset)$.

Variable forgetting is defined as $Forget(F, v) = (F[v \leftarrow false]) \vee (F[v \leftarrow true])$, with $F[v \leftarrow c]$ the formula obtained by substituting variable v by c in F [30, 13]. Projecting F on a set of variables $P \subseteq Var(F)$ (denoted by $Project(F, P)$) is equivalent to forgetting every variable in $Var(F) \setminus P$. By definition, we have $R_{Project(F,P)} = \{m \cap L \mid m \in R_F\}$ with $L = \{x, \neg x \mid x \in P\}$. We define a function $Split(F)$ that returns a set of variables $P \subseteq Var(F)$.

Definition 1. *Model Counting (# SAT) is the problem of computing the size of R_F.*

Definition 2. *Uniform random sampling (URS) is the problem of sampling a model from R_F such that every model $m \in R_F$ has probability $\frac{1}{|R_F|}$ of being sampled.*

4 DivKC

Our approach is based on the idea that splitting a formula F into smaller parts facilitates its compilation into a target language. More specifically, Zeyen et al. [33] show a strong correlation between the time and memory needed to compile a formula to d-DNNF and the number of variables and clauses in the formula. Our approach explicitly utilises this correlation by decomposing an input formula into subformulae with fewer variables and/or fewer clauses. A key strength of our approach, and one that distinguishes it from previous work, is that it does not rely on Shannon decomposition, which keeps the number of decompositions both predictable and small.

4.1 Overview of the Decomposition Algorithm

Algorithm 1 *Compile*(F)

Require: F is a satisfiable Boolean formula in CNF
1: $P \leftarrow Split(F)$
2: $G_P \leftarrow Project(F, P)$
3: $G_U \leftarrow \{c \in F \mid Var(c) \not\subseteq P\}$
4: **return** $ddnnf(G_P), ddnnf(G_U)$

A high-level description of our approach is shown in Algorithm 1. To compile a formula F into d-DNNF, we begin by applying a function *Split*(F), which returns a set of variables $P \subseteq$ *Var*(F) that will be used for projection (we present the algorithm to appropriately determine this subset P in Section 4.2). We then compute the projection of F onto P, yielding G_P. To do so, we use a resolution-based algorithm, which has been shown to be effective in [25, 13]. We compute G_U as the CNF resulting from the subset of clauses of F that have at least one variable not in P. Finally, we compile G_P and G_U in d-DNNF form using an off-the-shelf CNF to d-DNNF compiler. The main rationale behind our decomposition is that we isolate reasoning over the clauses strictly containing variables in P (via the compilation of G_P), whereas the other clauses are represented in G_U. The decomposition of F into G_P and G_U can later be exploited to design effective d-DNNF-based reasoning methods; We show how to exploit G_P and G_U for #SAT and URS in Sections 4.3 and 4.4, respectively. Before going into the details of these specific reasoning procedures, we demonstrate the structural and semantic properties of this decomposition.

Theorem 1. *Let* $\Gamma = \bigvee_{y \in R_{G_P}} ((G_U|_y) \wedge y)$ *and* $R_\Gamma = \bigcup_{y \in R_{G_P}} R_{(G_U|_y) \wedge y}$. *If* F *is satisfiable, then* $R_F = R_\Gamma$.

Proof. We sequentially prove $R_F \subseteq R_\Gamma$ and $R_\Gamma \subseteq R_F$.

$R_F \subseteq R_\Gamma$: Let $m \in R_F$. We know that $m \in R_{G_U}$ because $G_U \subseteq F$. By definition of *Project*(F, P) we know that $\exists y \in R_{G_P} : (y \subseteq m)$. Moreover, since R_Γ contains the models of $(G_U|_y) \wedge y$, we have that $m \in R_\Gamma$ and $R_F \subseteq R_\Gamma$.

$R_\Gamma \subseteq R_F$: Let $m \in R_\Gamma$. By definition of Γ we have $m \in R_{G_U}$. Let $G'_U = F \setminus G_U = \{c \in F \mid$ *Var*$(c) \subseteq P\}$. Hence, $R_F = R_{G_U} \cap R_{G'_U}$. By definition of *Project*(F, P), we have $G'_U \subseteq G_P$. In other words, for any complete assignment a to the variables in *Var*(F) we have $(\exists y \in R_{G_P} : y \subseteq a) \Rightarrow (a \in R_{G'_U})$. Since $m \in R_\Gamma$, there exists a $y \in R_{G_P}$ such that $m \models (G_U|_y) \wedge y$. Therefore, we have $m \in R_{G'_U}$ and $R_\Gamma \subseteq R_F$.

We conclude that $R_F = R_\Gamma$.

Theorem 2. *If* G_U *is in d-DNNF form then* $\Gamma = \bigvee_{y \in R_{G_P}} ((G_U|_y) \wedge y)$ *is in d-DNNF form.*

Proof. If $(G_U|_y) \wedge y$ is obtained by conditioning G_U on the literals in y, i.e., propagating the unit literals from y in G_U, then $(G_U|_y) \wedge y$ is a d-DNNF since conditioning a d-DNNF creates a new d-DNNF [7].

Let $a, b \in R_{G_P}$ such that $a \neq b$ then $\bigwedge_{l \in a} l \wedge \bigwedge_{l \in b} l$ is unsatisfiable because a and b are two distinct models of G_P. By definition, there exists at least one literal on which a and b disagree. Otherwise, the models would not be distinct.

Therefore, the main disjunction of Γ is deterministic, and Γ is indeed in d-DNNF form.

4.2 Choosing the Projection Set P

We decided to use hypergraph partitioning to choose a good projection set P. Other methods exist [2], but hypergraph partitioning offers a good balance be-

tween simplicity and efficiency [14]. While hypergraph partitioning is known to be NP-hard [16], similarly to SAT-solving, efficient solvers do exist [4]. Our approach is described in Algorithm 2.

To take advantage of hypergraph partitioning tools, we have to formulate our problem as a hypergraph partitioning problem. We construct the variable incidence graph (VIG) as follows. Each node n_v of the VIG is associated with a variable $v \in \mathit{Var}(F)$. Each clause of F (in CNF) is a hyperedge, i.e., for each clause $c \in F$ we construct a hyperedge that contains every node n_v such that $v \in \mathit{Var}(c)$.

We continue by running a hypergraph partitioning tool on the VIG of the formula F. The partitioning tool returns a function that associates each node n_v with a partition p. Partitioning tools try to create balanced partitions by cutting the smallest possible number of hyperedges. In our case, this means that the tool will partition the set of variables $\mathit{Var}(F)$ into subsets of roughly the same size. Moreover, the hypergraph partitioner will try to minimise the number of clauses expressed by using variables of different subsets. In other words, most clauses will be expressed within a single subset of the partition.

With our partition p computed, we can continue by computing our projection set P. We start by building a formula $\Delta = \{c \in F \mid \exists x, y \in c : (p(\mathit{Var}(x)) \neq p(\mathit{Var}(y)))\}$. The formula Δ contains every clause that connects at least two subsets of $\mathit{Var}(F)$ as defined by the partition p. We return $P = \mathit{Var}(\Delta)$.

Notice that G_U can be partitioned into subsets of clauses such that every subset has zero variables in common. In other words, G_U is built in such a way that a d-DNNF compiler can create a conjunction node early in the compilation process. An alternative is to form a partition $G_{U_1} \cup ... \cup G_{U_n} = G_U$ such that $\forall i, j : (i \neq j \Rightarrow (\mathit{Var}(G_{U_i}) \cap \mathit{Var}(G_{U_j}) = \emptyset))$. A consequence of this is that every component G_{U_i} can be compiled independently and, thus, in parallel.

Algorithm 2 *Split*(F)

Require: F is a Boolean formula in CNF
1: $vig \leftarrow \{\mathit{Var}(c) \mid c \in F\}$
2: $p \leftarrow \mathit{hypergraph\_partitioner}(vig)$
3: {p is a partition function, $p(var)$ tells us to which partition variable var belongs.}
4: $P \leftarrow \emptyset$
5: **for all** $c \in F$ **do**
6: **if** $\exists x, y \in c : (p(\mathit{Var}(x)) \neq p(\mathit{Var}(y)))$ **then**
7: $P \leftarrow P \cup \{\mathit{Var}(l) \mid l \in c\}$
8: **end if**
9: **end for**
10: {P contains the variables of every clause that connects multiple partitions.}
11: **return** P

4.3 Application to Model Counting

Direct Method Based on the G_P, G_U Decomposition. Our decomposition of F into G_P and G_U provides an immediate approach to model counting. We illustrate this approach in Figure 1. We consider $F = (a \vee b) \wedge (c \vee d) \wedge (a \vee c)$. Selecting $P = \{a, c\}$ yields $G_P = a \vee c$ and $G_U = (a \vee b) \wedge (c \vee d)$. By compiling G_P to a d-DNNF, we find $R_{G_P} = \{a \wedge c, a \wedge \neg c, \neg a \wedge c\}$. The resulting d-DNNF (according to Theorem 2) is shown graphically in Figure 1. By computing the sum of $|R_{(G_U|_y) \wedge y}|$ for every $y \in R_{G_P}$ we find the model count of F as indicated by Theorems 1 and 2.

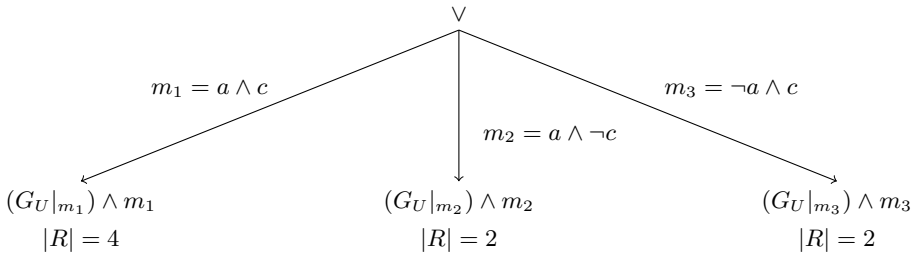

Fig. 1: The d-DNNF that we obtain by using Algorithm 1 to compile $F = (a \vee b) \wedge (c \vee d) \wedge (a \vee c)$ with $P = \{a, c\}$.

The issue with this direct approach is that $|R_{G_P}|$ is often huge, therefore prohibiting an exhaustive enumeration. To alleviate this, we use an optimisation shown by Lagniez and Lonca [12], which reduces the number of computations by enumerating orthogonal sufficient partial assignments of G_P instead of models of G_P. The number of orthogonal sufficient partial assignments of G_P is likely much smaller than the number of models $|R_{G_P}|$. Let a be such a sufficient partial assignment. Then a satisfies every clause in $G'_U = \{c \in F \mid \mathit{Var}(c) \subseteq P\}$. The remaining variables are unconstrained, and every model of $a \wedge G_U$ is a model of F (cf. Theorem 1). In other words, if we have a partial assignment a such that $|a| < |\mathit{Var}(G_P)|$ and G_P is satisfied by a (i.e., the variables not in a are unconstrained and G_P will evaluate to true under every assignment m such that $a \subseteq m$), then every model of $(G_U|_a) \wedge a$ is a model of F. Fortunately, d-DNNF compilation naturally produces orthogonal sufficient partial assignments. These can be extracted by considering the paths that originate at the root of the directed acyclic graph representing the d-DNNF of G_P. Each path corresponds to a sufficient partial assignment over the variables of G_P. These assignments are necessarily orthogonal due to the structural properties of d-DNNFs — specifically, the determinism condition ensures that no model y can be a superset of more than one path in the d-DNNF [12].

G_P and G_U as Lower and Upper Bounds for F. In case enumerating R_{G_P} and applying the direct method remains intractable due to large $|R_{G_P}|$, we demonstrate that our compilation process naturally generates lower and upper bounds to $|R_F|$.

Lemma 1. *G_P and G_U are such that $|R_{G_P}| \leq |R_F| \leq |R_{G_U}|$*

Proof. By construction, we have $\forall x \in R_{G_P} : \exists y \in R_F : (x \subseteq y)$ and $R_F \subseteq R_{G_U}$, therefore, $|R_{G_P}| \leq |R_F| \leq |R_{G_U}|$.

This implies that G_P and G_U can immediately be used to compute a sound interval for $|R_F|$. In practice, however, this interval can be too large (as confirmed in our experiments). This is why we next propose a statistical computation of tighter bounds.

Approximate Model Counting with Lower and Upper Bounds. Algorithm 3 shows how we approximately count the number of models of F given G_P and G_U. According to Theorem 1 and Theorem 2 we find that $|R_F| = \sum_{a \in R_{G_P}} |R_{(G_U|_a) \wedge a}|$. Instead of computing $|R_{(G_U|_a) \wedge a}|$ for every model a of G_P, we can estimate $\frac{|R_F|}{|R_{G_P}|}$, i.e., the average number of models that each model of G_P contributes to the total number of models. Suppose we have the multiset $A = \{|R_{(G_U|_a) \wedge a}| \mid a \in R_{G_P}\}$. If we sample uniformly at random from A then we have a random variable Y with expected value $\mathbb{E}[Y] = \sum_{Y_i \in A} Y_i \frac{1}{|R_{G_P}|} = \frac{|R_F|}{|R_{G_P}|}$. By linearity of expectation, we have $\mathbb{E}[|R_{G_P}| \times Y] = |R_F|$. Therefore, our algorithm, which approximates this expectation through sampling (see Algorithm 3), provides a consistent unbiased estimator of $|R_F|$ as a consequence of the law of large numbers.

Moreover, we use the central limit theorem [3] to construct asymptotic confidence intervals with confidence level $\alpha \in (0, 1)$, i.e., empirical intervals which will, in the limit $N \to \infty$, contain the true value of $|R_F|$ with probability at least $1 - \alpha$.

4.4 Application to Uniform Random Sampling

Directly sampling a model from F by sampling a model a from G_P and then returning a model from $(G_U|_a) \wedge a$ is unfortunately not uniform as we rarely have $\forall x, y \in R_{G_P} : (|R_{(G_U|_x) \wedge x}| = |R_{(G_U|_y) \wedge y}|)$. To this end, we propose to sample $S \subseteq R_{G_P}$ with $|S| = k$. We then sample uniformly from $R_T = \bigcup_{a \in S} R_{(G_U|_a) \wedge a}$ as demonstrated in Algorithm 4. When F is given in d-DNNF form, the procedure *random_model*(F) admits an efficient implementation based on the method of Sharma et al. [23]. In particular, they show that sampling from a d-DNNF is highly efficient.

Ideally, we would want our sampling algorithm to have a uniform probability of returning a model $m \in R_F$ (i.e., $P(m) = \frac{1}{|R_F|}$). While our algorithm does not allow for such guarantees (at least not most of the time with reasonable values for k), we can show that the value for $P(m)$ is bounded.

Algorithm 3 *AppMC*(G_P, G_U, α, N)

$v \leftarrow$ *empty array*
for all $1 \leq i \leq N$ **do**
  $a \leftarrow$ *random_model*(G_P)
  $Y_i \leftarrow |R_{(G_U|_a)\wedge a}| \times |R_{G_P}|$
  $v \leftarrow v \cup \{Y_i\}$
end for
$\overline{Y} \leftarrow$ *average*(v)
$S^2 \leftarrow \frac{\sum_{Y_i \in v}(Y_i - \overline{Y})^2}{N-1}$
$\sigma \leftarrow \frac{\sqrt{S^2}}{\sqrt{N}}$
$Y_l \leftarrow \overline{Y} - z_{\frac{\alpha}{2}}\sigma$
$Y_h \leftarrow \overline{Y} + z_{\frac{\alpha}{2}}\sigma$
return $\overline{Y}$, Y_l, Y_h

Lemma 2. *The probability $P(m)$ that Algorithm 4 returns model m is bounded as follows: $P_H \frac{\binom{|R_{G_P}|-1}{k-1}}{\binom{|R_{G_P}|}{k}} \leq P(m) \leq P_L \frac{\binom{|R_{G_P}|-1}{k-1}}{\binom{|R_{G_P}|}{k}}$, with $P_H = \frac{1}{\sum_{h\in H_k} h}$, $P_L = \frac{1}{\sum_{l\in L_k} l}$, and L_k (resp. H_k) the multiset containing the k smallest (resp. largest) values of $A = \{|R_{(G_U|_a)\wedge a}| \mid a \in G_P\}$.*

Proof. Suppose we have the multiset $A = \{|R_{(G_U|_a)\wedge a}| \mid a \in G_P\}$. Let L_k and H_k be the multisets containing the k smallest and k largest numbers of A, respectively. Notice that for any subset $S \subseteq R_{G_P}$, the sum of the respective model counts (i.e., $\sum_{a\in S} |R_{(G_U|_a)\wedge a}|$) is bounded by the sum of the elements in L_k and H_k: $\sum_{l\in L_k} l \leq \sum_{a\in S} |R_{(G_U|_a)\wedge a}| \leq \sum_{h\in H_k} h$. Therefore, if S is fixed, the probability of returning a model $m \in S$ is bounded as well: $P_H \leq P(m \mid S) \leq P_L$, with $P_H = \frac{1}{\sum_{h\in H_k} h}$ and $P_L = \frac{1}{\sum_{l\in L_k} l}$. We know that there are a total of $\binom{|R_{G_P}|}{k}$ different subsets of R_{G_P} of size k and that for $\binom{|R_{G_P}|-1}{k-1}$ of those ways, the model m is part of the selected subset of models (as a consequence of Theorem 2). Thus, the probability $P(m)$ is bounded as follows: $P_H \frac{\binom{|R_{G_P}|-1}{k-1}}{\binom{|R_{G_P}|}{k}} \leq P(m) \leq P_L \frac{\binom{|R_{G_P}|-1}{k-1}}{\binom{|R_{G_P}|}{k}}$.

We would like to add that if $k = |R_{G_P}|$, we have $P_H^{-1} = P_L^{-1} = |R_F|$ and thus our algorithm converges towards uniform random sampling with the limit $k \to |R_{G_P}|$.

5 Experimental Evaluation

5.1 Experimental Setup

We report on experiments assessing the benefits of DivKC. First, we demonstrate the ability of Algorithm 1 to compile into the d-DNNF form for challenging

Algorithm 4 *K-Sampler*(G_P, G_U, k, N)

```
R ← empty array
for all 1 ≤ i ≤ N do
    S ← subset of size k from R_{G_P}
    L ← empty array
    lmc ← 0
    for all a ∈ S do
        s_i ← |R_{(G_U|a)∧a}|
        L ← L ∪ {(a, s_i)}
        lmc ← lmc + s_i
    end for
    id ← random_number(1 ≤ r ≤ lmc)
    for all (a, s_i) ∈ L do
        if id ≤ s_i then
            R ← R ∪ {random_model((G_U|a) ∧ a)}
            break current loop
        else
            id ← id − s_i
        end if
    end for
end for
return R
```

formulae that D4 [14] was not able to compile. Thereby, we demonstrate that DivKC can complement D4 and enhance its knowledge compilation capability, making it compile formulae it could not without DivKC.

Second, we want to show the benefits of DivKC in an application to #SAT. We aim to demonstrate the quality of the confidence intervals produced by Algorithm 3 in terms of coverage and precision. For coverage, we measure the percentage of formulae for which the true model count lies within the interval (i.e., how often we have $|R_F| \in [Y_l; Y_h]$). For precision, we compare the interval with that of the lower and upper bounds produced by Lemma 1 (which guarantees 100% coverage by construction). We measure the percentage of formulae for which the confidence interval is included in the interval formed by Lemma 1's lower and upper bounds, and in these cases, we measure the relative size of the two intervals.

Third, we evaluate the ability of DivKC to generate uniform random samples for various formulae. For this, we generate samples using Algorithm 4 and execute the uniformity test suite proposed by Zeyen et al. [34].

Datasets. To test the compilation and model counting abilities of DivKC, we collected multiple datasets from Lagniez and Marquis [14], Soos [24], Sundermann et al. [26], and Plazar et al. [20]. For the sake of fine-grained traceability, we keep these datasets separated in our result tables. Detailed tables are provided in our companion GitHub repository [32]. Table 1 presents the characteristics of the formulae included in these datasets. Each line represents a specific dataset.

The line marked as global groups the datasets into one. Each subsequent line represents a different dataset.

The Lagniez and Marquis [14] dataset is a diverse dataset containing 1979 formulae. The dataset contains diverse problems ranging from Bayesian networks to digital circuits and configuration. This dataset also contains handmade and random formulae.

The Soos [24] dataset contains 1896 formulae from various sources, including the Model Counting Competitions.

The Sundermann et al. [26] dataset consists of 278 formulae, most of which come from the configurable software domain. The dataset contains multiple versions and variants of each formula. To avoid having too many similar formulae, we restricted our experiments to the most recent version and variant of each formula.

The Plazar et al. [20] dataset contains 503 formulae consisting of a feature model benchmark as well as other formulae collected from [8].

Dataset	$\#F_{total}$	min(|*Var*(*F*)|)	max(|*Var*(*F*)|)	min(|*F*|)	max(|*F*|)
Global	4656	0	8286433	0	7689680
↪ Lagniez [14]	1979	5	229100	10	399794
↪ Soos [24]	1896	2	8286433	1	7689680
↪ Plazar [20]	503	14	486193	31	2598178
↪ Sundermann [26]	278	0	31012	0	350120

Table 1: Dataset summary. The first column indicates the dataset, and the #F column indicates how many formulae the dataset contains. The following columns indicate the minimum and maximum number of variables (resp. clauses) in the dataset.

To test the uniformity of our approach, we used the same dataset as used by Zeyen et al. [34] to allow for a direct comparison with their results. Using the same dataset allows us to use the results in [34] as a baseline and use our results to construct a direct comparison between the samplers tested by Zeyen et al. [34] and our proposed sampler.

Infrastructure. The experiments were computed on an HPC containing 354 nodes, each of which has 256 GB of RAM and 2 AMD Epyc ROME 7H12 CPUs running at 2.6 GHz.

Computation Budget. We set the following computational budget for the evaluated approaches. Compiling a formula with D4 [14] is limited to 64GB of memory and five hours of computation for each formula. Algorithm 1 is limited to 30 minutes for the splitting procedure, two hours for the computation of the projection G_P (both within 64GB of memory), and one hour and 16GB of memory to compile each component (G_P and G_U) by using D4 [14]. Therefore,

Algorithm 1 is given a total of three hours and 30 minutes, considering that compiling G_P and G_U can be done in parallel. Our approximate model counting procedure (Algorithm 3) is limited to one and a half hours of computation and 64GB of memory. Therefore, our approach to approximate model counting is limited to five hours of computation (including the compilation phase), which is the same limit given to D4 [14].

5.2 Experimental Results

Dataset	$\#F_{total}$	$\#\text{D4} \wedge \neg\text{DivKC}$	$\#\neg\text{D4}$	$\#\text{DivKC} \wedge \neg\text{D4}$
Global	4656	643	672	**114**
↪ Lagniez [14]	1979	256	214	**54**
↪ Soos [24]	1896	311	394	**54**
↪ Plazar [20]	503	58	61	**6**
↪ Sundermann [26]	278	18	3	0

Table 2: Experimental results regarding the scalability of Algorithm 1. Column $\#F_{total}$ indicates the total number of formulae in each dataset. The next column shows the number of formulae compiled only by D4 [14] but not by Algorithm 1. Column $\#\neg$D4 shows the number of formulae not compiled by D4. The last column indicates the number of formulae that were only compiled by Algorithm 1, but not by D4.

Knowledge Compilation. We compare DivKC with D4 to evaluate the compilation capabilities of our approach. The results for our compilation algorithm are shown in Table 2. The $\#F_{total}$ column indicates the total number of formulae in each dataset. The main columns for our evaluation are $\#\neg$D4 and $\#\text{DivKC} \wedge \neg\text{D4}$. The former shows how many formulae could not be compiled with D4 within our computational budget (64GB of memory and five hours), while the latter shows how many of these were successfully compiled by our approach.

Our main result shows that, among the 672 formulae that D4 failed to compile initially within our computational budget and given our current hardware, 114 can be successfully compiled using our DivKC approach, with D4 serving as the backbone d-DNNF compiler.

Approximate Model Counting. To evaluate Algorithm 3, we start by measuring the coverage (Table 3) and precision (Table 4) of the intervals constructed by our approach (with parameters $\alpha = 0.01$ and $N = 10{,}000$). Coverage reflects how often the returned bounds contain the true model count, thereby indicating the accuracy of Algorithm 3. We constrained the computational budget to 64GB of memory and a maximum runtime of 1.5 hours per execution of Algorithm 3.

Dataset	#F	$Y_l \leq \lvert R_F \rvert$	$Y_h \geq \lvert R_F \rvert$	Coverage	$\lvert R_{G_P} \rvert \leq \lvert R_F \rvert \leq \lvert R_{G_U} \rvert$
Global	3341	0.988	0.869	0.857	1.000
↪ Lagniez [14]	1509	0.993	0.914	0.907	1.000
↪ Soos [24]	1191	0.981	0.915	0.896	1.000
↪ Plazar [20]	384	0.987	0.815	0.802	1.000
↪ Sundermann [26]	257	0.996	0.475	0.471	1.000

Table 3: Experimental results for Algorithm 3. Column #F indicates with how many formulae the statistics have been computed. The 'Coverage' column indicates how often $|R_F|$ is within the confidence interval $[Y_l; Y_h]$ and thus measures the accuracy of our method.

In Table 3, the column #F shows for how many formulae we managed to compute both the exact model count with D4 [14] (within the same computational budget as mentioned above) and our approximate model counter. The following columns indicate how often the returned lower bound was smaller than the true model count ($Y_l \leq |R_F|$), and how often the returned upper bound was larger than the true model count ($Y_h \geq |R_F|$). The 'Coverage' column indicates how often $|R_F|$ is within the confidence interval $[Y_l; Y_h]$ and thus measures the accuracy of our method. The last column serves as a sanity check for the bounds obtained by using Lemma 1. We observe that the bounds obtained by using Lemma 1 are as expected.

We begin our evaluation of Algorithm 3 by discussing its coverage performance. For 98% of all formulae, our approach correctly calculates a lower bound to the total number of models. However, the upper bound is only correct in 86% of the cases, showing that our approach tends to underestimate the model count of the formula. Similarly, our approach correctly computes a lower bound in 100% of the cases for the feature model subset of the Plazar et al. [20] dataset (as can be seen on our companion GitHub [32]) and in 99% of the cases for the Sundermann et al. [26] dataset. However, the upper bound is only correct in 47% of the cases. The Sundermann et al. [26] dataset mostly contains feature models of software systems. Coverage is lower than expected for the Sundermann et al. [26] dataset because our approach converges only asymptotically under the central limit theorem [3] and law of large numbers, and limiting computations to 10,000 samples per formula can prevent full convergence. Therefore, we deduce that our approach underestimates the number of models of a feature model. Over all datasets, both the upper and lower bounds are correct in 85% of the cases, and the lower bounds have an experimental reliability of 98%, showing the accuracy of our approximate method.

To evaluate the precision of Algorithm 3, we compare its computed bounds with those obtained using Lemma 1. Table 4 relates the upper and lower bounds computed by Algorithm 3 with the bounds obtained by using Lemma 1. As above, the #F column indicates the number of formulae on which the following statistics have been computed. The third column indicates how often the lower bound computed by Algorithm 3 is greater than the lower bound obtained

Dataset	#F	$Y_l \geq \lvert R_{G_P} \rvert$	$Y_h \leq \lvert R_{G_U} \rvert$	Both	$median(r_c)$	$max(r_c)$
Global	3341	0.834	0.869	0.752	9.58e-9	1.0
↪ Lagniez [14]	1509	0.868	0.913	0.801	7.53e-9	1.0
↪ Soos [24]	1191	0.940	0.915	0.863	2.55e-7	0.085
↪ Plazar [20]	384	0.763	0.815	0.656	6.61e-14	0.0137
↪ Sundermann [26]	257	0.249	0.471	0.089	1.6e-13	0.126

Table 4: Experimental results comparing the bounds obtained with Algorithm 3 and with Lemma 1. Column #F indicates with how many formulae the statistics have been computed. The 'Both' column indicates how often we have $Y_l \geq |R_{G_P}| \wedge Y_l \leq |R_F|$ and $Y_h \leq |R_{G_U}| \wedge Y_h \geq |R_F|$. The last two columns represent the observed median and maximum values of the ratio $r_c = \frac{min(Y_h, |R_{G_U}|) - max(Y_l, |R_{G_P}|)}{|R_{G_U}| - |R_{G_P}|}$, which was calculated exclusively if $Y_l \leq |R_F| \leq Y_h$. The number of formulae on which the last two columns are computed can easily be obtained by multiplying the #F column with the 'Coverage' column in Table 3.

through Lemma 1 and smaller than the true model count ($Y_l \geq |R_{G_P}| \wedge Y_l \leq |R_F|$). The fourth column indicates a similar result but for the upper bound ($Y_h \leq |R_{G_U}| \wedge Y_h \geq |R_F|$). In other words, these two columns show how often Algorithm 3 gives us bounds that are better than or equal to those provided by Lemma 1. The 'Both' column indicates how often the bounds returned by Algorithm 3 were both correct and better than the bounds obtained with Lemma 1. The last two columns indicate the median and maximum value for the ratio $r_c = \frac{min(Y_h, |R_{G_U}|) - max(Y_l, |R_{G_P}|)}{|R_{G_U}| - |R_{G_P}|}$, which was calculated exclusively for the bounds that meet the predicate $Y_l \leq |R_F| \leq Y_h$ (the number obtained by multiplying the #F column in this table with the 'Coverage' column in Table 3). The r_c ratio quantifies the difference between the bounds obtained by using Lemma 1 and the bounds obtained by using Algorithm 3. To compute r_c we use $min(Y_h, |R_{G_U}|)$ and $max(Y_l, |R_{G_P}|)$ because detecting that the bounds returned by Algorithm 3 are worse than the bounds obtained with Lemma 1 is easy, and therefore, a user can easily use the better bounds.

We observe that in general, Algorithm 3 provides tighter bounds than Lemma 1 in 75% of the cases. As previously noted, the results vary depending on the dataset. As an example, Algorithm 3 performs poorly on the Sundermann et al. [26] dataset and on the feature model subset of the Plazar et al. [20] dataset. On the other hand, Algorithm 3 performs well on the Bayesian network subset of the Lagniez and Marquis [14] dataset, as we observe a success rate of 86%. Moreover, we find that the bounds returned by Algorithm 3 are overall much tighter than the bounds obtained by using Lemma 1 as the global median value for $r_c = 9.5 \times 10^{-9}$.

To complete our evaluation of Algorithm 3, we compare it against ApproxMC 7 [21] because it is generally considered to be the state-of-the-art in approximate model counting. Moreover, ApproxMC has strong theoretical guarantees.

Dataset	#F	$l \leq Y_{\text{ApproxMC}} \leq h$	$l \leq Y \leq h$
Global	2782	0.977	0.881
↪ Lagniez [14]	1386	1.000	0.882
↪ Soos [24]	1180	0.970	0.873
↪ Plazar [20]	203	0.862	0.921
↪ Sundermann [26]	13	1.000	0.846

Table 5: Experimental results comparing the accuracy of Algorithm 3 with ApproxMC 7. Column #F indicates with how many formulae the statistics have been computed. Column $l \leq Y_{\text{ApproxMC}} \leq h$ (resp. $l \leq Y \leq h$) indicates how often the model count returned by ApproxMC 7 (resp. Algorithm 3) is within the indicated bounds, with $l = \frac{|R_F|}{1.2}$ and $h = 1.2 \times |R_F|$.

Dataset	#F	#DivKC	$log_{10}(min)$	$mean$	$median$	$log_{10}(max)$
Global	2888	2265	-3.2	124.5	4.8	5.3
↪ Lagniez [14]	1436	1179	-2.9	80.5	5.5	4.4
↪ Soos [24]	1235	953	-3.0	29.6	4.5	4.4
↪ Plazar [20]	204	122	-3.2	2.7	1.7	1.5
↪ Sundermann [26]	13	11	-0.6	15904.5	29.6	5.3

Table 6: Experimental results comparing the runtime of Algorithm 3 with ApproxMC 7 for 20 runs. Column #F indicates the number of formulae over which the statistics were computed, and column #DivKC shows how often DivKC was faster than ApproxMC 7. The remaining columns report how much faster DivKC was, based on the logarithm of the minimum, the mean, the median, and the logarithm of the maximum of the ratio: ApproxMC 7 execution time divided by DivKC execution time.

The results of this comparison are presented in Tables 5 and 6, which highlight differences in accuracy and runtime performance.

Table 5 presents the accuracy results. Column #F reports the number of formulae for which the statistics were computed from the results produced by ApproxMC 7 and Algorithm 3. Thus, #F indicates the number of formulae on which we successfully ran ApproxMC 7, Algorithm 3, and D4. Column $l \leq Y_{\text{ApproxMC}} \leq h$ (resp. $l \leq Y \leq h$) reports how often the model count returned by ApproxMC 7 (resp. Algorithm 3) falls within the specified bounds, with $l = \frac{|R_F|}{1.2}$ and $h = 1.2 \times |R_F|$.

The bounds we use follow the definition of a probably approximately correct (PAC) counter from [21], where a probabilistic algorithm, given parameters δ and ε, returns an estimate Y such that $P(l \leq Y \leq h) \geq 1 - \delta$, with $l = \frac{|R_F|}{1+\varepsilon}$ and $h = (1 + \varepsilon)|R_F|$. In our experiments, we set $\varepsilon = 0.2$ and $\delta = 0.1$ as input parameters to ApproxMC. The value of ε matches the value used by Pote et al. [21], while $\delta = 0.1$ is a standard choice in statistical settings. Additionally, we modified Algorithm 3 to use at most 10,000 samples, and to terminate early if $(Y_l \geq \frac{Y}{1.1}) \wedge (Y_h \leq 1.1Y)$.

Overall, ApproxMC demonstrates better accuracy, returning estimates within bounds in approximately 98% of cases, compared to 88% for Algorithm 3. While this indicates a performance gap, 88% still reflects a reasonably high level of accuracy, especially considering the added benefit of the reusable data structure produced by DivKC. This may make DivKC particularly appealing in scenarios where repeated queries are expected. Therefore, ApproxMC may not always be the most practical choice despite its higher accuracy.

Table 6 compares the runtimes of ApproxMC and Algorithm 3. To compare both algorithms, we simulate 20 runs of both approaches per formula (to limit the computational budget). For ApproxMC, we performed 4 actual runs per formula and multiplied the total runtime by 5. For Algorithm 3, we ran the compilation phase once and then executed the algorithm 4 times per formula; the total runtime of Algorithm 3 was also multiplied by 5 to simulate 20 runs. The reported execution time for Algorithm 3 includes one call to DivKC and 20 simulated runs of Algorithm 3. ApproxMC had a computational budget of five hours and 64GB of memory per run.

Column #F indicates the number of formulae over which the statistics were computed, and column #DivKC shows how often DivKC was faster than ApproxMC. Thus, #F indicates the number of formulae on which we successfully ran ApproxMC 7 and Algorithm 3. The remaining columns report how much faster DivKC was, based on the logarithm of the minimum, the mean, the median, and the logarithm of the maximum of the ratio: ApproxMC execution time divided by DivKC execution time.

We observe that DivKC was faster than ApproxMC in approximately 78% of the cases. The speedup was substantial — on average, 124.5 times faster, with a median speedup of 4.8. In the most extreme case, DivKC was up to $197,602.8$ times faster.

Therefore, while Algorithm 3 may not be as reliable as ApproxMC in terms of accuracy, it offers significant performance advantages. Moreover, the reusable data structure generated by DivKC can make it particularly useful in scenarios where repeated model counting is required.

Approximate Uniform Random Sampling. We used the test suite and dataset proposed by Zeyen et al. [34] to test our heuristic-based sampler presented in Algorithm 4. Algorithm 4 is run with $k = 50$ and with batch sizes of $N = 1000$, similarly to the experimental setup in [34]. We used the same dataset (which the authors called the Ω dataset) to facilitate the comparison with their results. The authors proposed five tests, of which we used only four, as the last test faces scalability issues according to the authors' original experiments. The simplest test is the modbit test. Table 7 shows a reproduction of the results in [34]. The table contains the results for heuristic-based samplers. The results for UniGen3 are provided as a baseline because UniGen3 comes with strong theoretical guarantees of uniformity (at the cost of performance [20]) and is therefore out of scope. We extended the table with our results for Algorithm 4. With our sampler, we obtained a Harmonic mean p-value of 0.13 for the modbit

	Modbit (q = 8)		VF		Birthday		SFpC	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value
K-Sampler	178	**0.129**	176	0.000	140	0.005	77	0.000
QuickSampler	188	0.000	186	0.000	139	0.000	77	0.000
STS	193	0.000	191	0.000	138	0.000	81	0.000
CMSGen	144	0.000	143	0.000	93	0.000	71	0.000
UniGen3	192	**0.234**	183	**0.083**	130	**0.274**	76	**0.253**

Table 7: Experimental results for the Ω dataset introduced in [34] and extended with our results for Algorithm 4. For each test (and for each formula), each sampler was called multiple times to generate samples of size 1000. The bold p-values are all greater than our significance level $\alpha = 0.01$. #F indicates the number of formulae on which the test was successfully performed (i.e., without timeouts or out-of-memory errors).

test with $q = 8$, which indicates that our sampler passed the modbit ($q = 8$) test. Our sampler also passed the modbit test with $q = 2$ and $q = 4$.

Our sampler is already an improvement, as none of the heuristic-based samplers (i.e., samplers with no guarantees of uniformity) tested in [34] passed a single test on the dataset. For the VF, SFpC, and Birthday tests, we obtained a harmonic mean p-value of 0. Given that the harmonic mean p-value is below the usual threshold value ($\alpha = 0.01$), we conclude that Algorithm 4 fails the VF, SFpC, and Birthday tests. However, the Birthday test also indicates the number of observed repetitions, i.e., the number of times the sampler under test returned the same model. This is interesting information as it indicates whether a sampler often returns the same model or if it seldom returns the same model. Frequent repetition may indicate poor exploration of the model space, which can be problematic. The Birthday test is also the only test proposed by [34] which allows for a finer quantitative analysis.

In our experiments, we set the expected number of duplicates for the birthday test to 10, exactly like in [34]. By doing so, we obtain results that are comparable with the results in [34]. Table 8 shows a reproduction of the results in [34]. We extended the table with our results for Algorithm 4.

Discussing the quantitative details of the failed tests, we still observe that our approach brings improvements over the other heuristic-based URS approaches as it generates much fewer repeats than QuickSampler [8] and CMSGen [10]. Moreover, we find that our results are competitive with other URS approaches as the average number of observed repeats is only off by 10% when comparing with UniGen3, a uniform random sampler which offers theoretical guarantees of uniformity.

Overall, our heuristic-based sampler **passes more tests than any other** heuristic-based sampler that is tested by Zeyen et al. [34].

	Uniformity		Observed number of repetitions			
Sampler	#F	p-value	min	max	average	median
K-Sampler	140	0.005	3	26	10.75	10
QuickSampler	139	0.000	0	29858	480.01	4
STS	138	0.000	0	27	5.20	4
CMSGen	93	0.000	5	12846	991.37	33
UniGen3	130	**0.274**	3	18	9.78	10

Table 8: Extended experimental results for the birthday test with the Ω dataset introduced in [34] and extended with our results for Algorithm 4. The bold p-values are all greater than our significance level $\alpha = 0.01$. #F indicates the number of formulae on which the test was successfully performed (i.e., without timeouts or out-of-memory errors).

6 Conclusion

In this paper, we developed DivKC, a divide-and-conquer method to split a formula into components that can then be compiled independently to d-DNNF. By using Theorems 1 and 2, we obtain a d-DNNF that is equivalent to the original formula. Our experiments demonstrated that DivKC compiles 114 formulae that were previously out of reach for the state-of-the-art D4 [14] compiler. We also explored two other applications of DivKC. First, we designed an approximate model counter that comes with statistical guarantees. While this new model counter accurately estimates 85% of the formulae, it struggles with formulae coming from feature models. Second, we exploited DivKC to build a heuristic-based uniform random sampler.

This heuristic-based sampler is the first heuristic-based sampler to validate at least one test of the test suite proposed in [34]. This paves the way to the design of novel quasi-uniform samplers, which are of interest for many practical applications, knowing that truly uniform samplers do not scale well [20]. For future work, we plan to explore alternative split heuristics to improve the efficiency and generality of our approach. Another promising direction is to investigate further how the upper bound used in our method could be exploited.

Acknowledgements

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR). Gilles Perrouin is a FNRS Research Associate.

Maxime Cordy and Olivier Zeyen are supported by FNR Luxembourg (grants C23/IS/18177547/VARIANCE and AFR Grant 17047437).

References

[1] D. Achlioptas, Zayd Hammoudeh, and P. Theodoropoulos. “Fast Sampling of Perfectly Uniform Satisfying Assignments”. In: *SAT*. 2018 (cit. on p. 3).

[2] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. "The Community Structure of SAT Formulas". In: *International Conference on Theory and Applications of Satisfiability Testing*. 2012 (cit. on p. 5).

[3] Robert B Ash and Catherine A Doléans-Dade. *Probability and measure theory*. Academic press, 2000 (cit. on pp. 8, 13).

[4] Ümit V Çatalyürek and Cevdet Aykanat. *PaToH (Partitioning Tool for Hypergraphs)*. 2011 (cit. on p. 6).

[5] Adnan Darwiche et al. "New advances in compiling CNF to decomposable negation normal form". In: *Proc. of ECAI*. Citeseer. 2004, pp. 328–332 (cit. on p. 3).

[6] Adnan Darwiche. "On the tractable counting of theory models and its application to belief revision and truth maintenance". In: *arXiv preprint cs/0003044* (2000) (cit. on p. 1).

[7] Adnan Darwiche and Pierre Marquis. "A Knowledge Compilation Map". In: *J. Artif. Intell. Res.* 17 (2002), pp. 229–264 (cit. on pp. 1, 5).

[8] Rafael Dutra et al. "Efficient sampling of SAT solutions for testing". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 2018, pp. 549–559. DOI: 10.1145/3180155.3180248. URL: http://doi.acm.org/10.1145/3180155.3180248 (cit. on pp. 3, 11, 17).

[9] Stefano Ermon, Carla Gomes, and Bart Selman. "Uniform Solution Sampling Using a Constraint Solver as an Oracle". In: *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*. UAI'12. Catalina Island, CA: AUAI Press, 2012, pp. 255–264. ISBN: 9780974903989 (cit. on p. 3).

[10] Priyanka Golia et al. "Designing Samplers is Easy: The Boon of Testers". In: *2021 Formal Methods in Computer Aided Design (FMCAD)* (2021), pp. 222–230 (cit. on p. 17).

[11] Carla P Gomes et al. "From Sampling to Model Counting." In: *IJCAI*. Vol. 2007. 2007, pp. 2293–2299 (cit. on p. 2).

[12] Jean-Marie Lagniez and Emmanuel Lonca. "Leveraging decision-DNNF compilation for enumerating disjoint partial models". In: *21st International Conference on Principles of Knowledge Representation and Reasoning (KR 2024)*. 2024 (cit. on p. 7).

[13] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. "Improving Model Counting by Leveraging Definability". In: *International Joint Conference on Artificial Intelligence*. 2016. URL: https://api.semanticscholar.org/CorpusID:6303269 (cit. on pp. 4, 5).

[14] Jean-Marie Lagniez and Pierre Marquis. "An Improved Decision-DNNF Compiler". In: *IJCAI*. 2017 (cit. on pp. 1–3, 6, 10–15, 18).

[15] Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. "DMC: a distributed model counter". In: *27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. 2018, pp. 1331–1338 (cit. on p. 3).

[16] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012 (cit. on p. 6).

[17] Christian Muise et al. "D sharp: fast d-DNNF compilation with sharpSAT". In: *Advances in Artificial Intelligence: 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings 25*. Springer. 2012, pp. 356–361 (cit. on p. 3).

[18] Jeho Oh et al. "Finding near-optimal configurations in product lines by random sampling". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. Ed. by Eric Bodden et al. ACM, 2017, pp. 61–71. DOI: 10.1145/3106237.3106273. URL: https://doi.org/10.1145/3106237.3106273 (cit. on p. 1).

[19] Jeho Oh et al. *Scalable Uniform Sampling for Real-World Software Product Lines*. Tech. rep. TR-20-01. 2020 (cit. on p. 3).

[20] Quentin Plazar et al. "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?" In: *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. 2019, pp. 240–251 (cit. on pp. 10–16, 18).

[21] Yash Pote, Kuldeep S Meel, and Jiong Yang. "Towards Real-Time Approximate Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. 11. 2025, pp. 11318–11326 (cit. on pp. 1, 2, 14, 15).

[22] Shubham Sharma et al. "GANAK: A Scalable Probabilistic Exact Model Counter." In: *IJCAI*. Vol. 19. 2019. 2019, pp. 1169–1176 (cit. on p. 2).

[23] Shubham Sharma et al. "Knowledge Compilation meets Uniform Sampling". In: *Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Nov. 2018 (cit. on pp. 1, 3, 8).

[24] Mate Soos. *Benchmarks used for Approximate Model Counting*. Zenodo, Jan. 2024. DOI: 10.5281/zenodo.10449477. URL: https://doi.org/10.5281/zenodo.10449477 (cit. on pp. 10–15).

[25] Mate Soos and Kuldeep S Meel. "Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling". In: *arXiv preprint arXiv:2110.09026* (2021) (cit. on p. 5).

[26] Chico Sundermann et al. "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking". In: *Proceedings of the 28th ACM International Systems and Software Product Line Conference*. New York, NY, USA: ACM, Sept. 2024, pp. 54–65 (cit. on pp. 10–15).

[27] Chico Sundermann et al. "Evaluating state-of-the-art # SAT solvers on industrial configuration spaces". In: *Empirical Software Engineering* 28 (2023) (cit. on pp. 1, 3).

[28] Chico Sundermann et al. "Reusing d-DNNFs for Efficient Feature-Model Counting". In: *ACM Transactions on Software Engineering and Methodology* 33.8 (2024), pp. 1–32 (cit. on pp. 1–3).

[29] Marc Thurley. "sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP". In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–429. ISBN: 978-3-540-37207-3 (cit. on p. 2).

[30] Yisong Wang. "On Forgetting in Tractable Propositional Fragments". In: *ArXiv* abs/1502.02799 (2015). URL: `https://api.semanticscholar.org/CorpusID:6588613` (cit. on p. 4).

[31] Wei Wei and Bart Selman. "A new approach to model counting". In: *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, UK, June 19-23, 2005. Proceedings 8*. Springer. 2005, pp. 324–339 (cit. on p. 2).

[32] Olivier Zeyen. *DivKC: A Divide-and-Conquer Approach to Knowledge Compilation*. `https://github.com/serval-uni-lu/divkc`. Zenodo archive: `https://doi.org/10.5281/zenodo.18097437`. 2025 (cit. on pp. 2, 10, 13).

[33] Olivier Zeyen et al. "Preprocessing is What You Need: Understanding and Predicting the Complexity of SAT-based Uniform Random Sampling". In: *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 2024, pp. 23–32 (cit. on p. 4).

[34] Olivier Zeyen et al. "Testing Uniform Random Samplers: Methods, Datasets and Protocols". In: *ACM Transactions on Software Engineering and Methodology* (2025) (cit. on pp. 2, 10, 11, 16–18).