PhD-FSTM-2025-100
Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 23 September 2025 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

Olivier Georges Rémy Zeyen
Born on 29 April 1997 in Luxembourg (Luxembourg)

Scaling up Uniform Random Sampling

Dissertation defence committee

Dr. Maxime Cordy, Dissertation Supervisor
Assistant Professor, Université du Luxembourg, Luxembourg

Dr. Michail Papadakis, Chairman
Associate Professor, Université du Luxembourg, Luxembourg

Dr. Nicolas Navet
Professor, Université du Luxembourg, Luxembourg

Dr. Jean-Marie Lagniez
Professor, CRIL - Université d'Artois, France

Dr. Gilles Perrouin
Research associate, Université de Namur, Belgium

Abstract

Despite its NP-completeness, the Boolean satisfiability problem (SAT) gave birth to highly efficient tools that can find solutions to a Boolean formula. Boolean formulae can succinctly represent vast, constrained search spaces, such as the configuration options of variability-intensive systems like the Linux kernel. Due to the sheer size of these spaces, exhaustive exploration is typically infeasible. As a result, most testing approaches rely on sampling a subset of solutions for analysis. A desirable property of such samples is *uniformity*: each solution should get the same selection probability. This property motivated the design of uniform random samplers, relying on SAT solvers or model counters, and achieving different trade-offs between uniformity and scalability.

In this thesis, we contribute to a deeper understanding of uniform random sampling complexity through several advances. First, we introduce a novel, efficient parallel algorithm to compute the number of equivalence classes in Boolean formulae, a structural metric that strongly correlates with sampling time and memory usage. Leveraging these correlations, we develop a classifier that accurately predicts whether sampling will exceed computational budgets.

Second, we deepen our understanding of sampling complexity by exploring synthetic formulae and phase transitions. We investigate how phase transitions can explain the practical complexity of sampling. Our results, computed on 11,409 synthetic formulae and 4656 real-world formulae, show that phase transitions occur in both uniform random sampling and SAT-solving, but at a different clause-to-variable ratio than for SAT tasks. We further reveal that low formula modularity is correlated with a higher uniform random sampling time. Overall, our work contributes to a principled understanding of uniform random sampling complexity.

Third, we develop a statistical testing framework to support the evaluation and development of new uniform random sampling algorithms. While assessing the uniformity of a sampler shares similarities with testing pseudo-random number generators (PRNGs), key differences make the problem more challenging: sampling is significantly slower, and the space of valid solutions is often highly constrained by the input formula. As a result, traditional PRNG testing methods are insufficient for this domain. To address this, we introduce a suite of five statistical tests specifically

designed for evaluating uniformity in constrained sampling scenarios. We apply these tests to seven existing samplers, showing their effectiveness and diagnostic power. Additionally, we demonstrate the influence of the Boolean formula given as input to the samplers under test on the test results.

Finally, we introduce a divide-and-conquer approach to knowledge compilation (KC). At the time of writing, knowledge compilation is one of the most effective methods to achieve uniform random sampling. However, KC still fails to scale on some Boolean formulae, including some representing the variability of large configurable systems. Concretely, our DivKC algorithm decomposes 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.

Acknowledgments

I have received an incredible amount of support from many people throughout my PhD journey, and I would like to take this opportunity to thank them all.

First and foremost, I would like to express my sincere gratitude to Dr. Maxime Cordy, whose guidance and mentorship have been instrumental in shaping my academic development. His thoughtful feedback and patience made this challenging journey possible. I am truly thankful for the time and effort he invested in helping me grow as a researcher.

I would also like to thank all of my co-authors and collaborators, whose insights and critical feedback helped strengthen my work. Engaging in discussions with you was an invaluable source of learning.

I am grateful to the Luxembourg National Research Fund (FNR) for financially supporting my PhD through the AFR grant. This support made my research possible and allowed me to pursue questions I deeply care about.

I want to thank my colleagues and friends, both inside and outside the lab, who contributed to the human side of this experience. Your companionship and occasional distractions made even the toughest moments easier to handle. Whether through shared coffee breaks, academic debates, or simply being there, you brought joy and balance to this journey.

I am deeply thankful to my family for their unwavering support and understanding over the years.

Finally, I want to express my deepest gratitude to my partner, Melissa Giacometti, whose love, patience, and steadfast support carried me through the most difficult moments. Your faith in me never wavered, and your encouragement gave me the strength to keep going when I needed it most.

Contents

1

Introduction

This chapter discusses the context and difficulties associated with uniform random sampling. We start by introducing uniform random sampling with some applications. We follow by discussing the issues related to the scalability of uniform random sampling. Finally, we summarise the contributions of this dissertation.

Contents

1.1 Context

Uniform Random Sampling (URS) is the problem of generating random SAT-solutions from a Boolean formula, such that every solution of the input formula gets a uniform probability of being returned. URS is a problem of both theoretical and practical interest.

URS is especially relevant in software engineering, where configuration spaces grow exponentially, making exhaustive testing infeasible. For instance, software product lines (SPLs) often have huge configuration spaces due to combinatorial explosion. Thus, to test an SPL, testing every configuration is often intractable even for small numbers of features. For example, an older version of JHipster [HNA+18] has 45 features, which result in 26256 possible configurations. Testing every configuration would be very time-consuming or even unfeasible, depending on the available computation budget. URS provides a practical solution: sample a subset of configurations uniformly at random, allowing testing to scale with available computational resources while preserving statistical soundness. Adjusting the sample size allows practitioners to balance test coverage and confidence with computational cost [OBM+17; OGB19; PAP+19].

Similarly, Oh et al. [OBM+17] showed that URS can be used to search for optimal configurations. The authors use URS to guide a recursive search of the configuration space of SPLs to directly find near-optimal configurations instead of constructing a prediction model. The authors compared their approach to prior approaches focused on training and using a performance model. They showed that their use of URS is simpler and has better accuracy and efficiency than other approaches.

Other applications include deep learning verification, where inputs are drawn from an unknown distribution [BCM+21] or evolutionary algorithms, where de Perthuis de Laillevault et al. [dPDD15] theoretically demonstrated the relevance of repeated uniform random sampling when initialising populations. Improving URS thus has a broad interdisciplinary impact.

When evaluating URS techniques (or *samplers*), two quality criteria matter: *uniformity* and *scalability*. Uniformity evaluates how close the distribution of the sampled solutions is to the uniform distribution. Uniformity is a central property that ensures a fair and unbiased representation of the entire solution space. This unbiasedness is crucial for reliable statistical estimates, robust software testing, and sound decision-making in applications ranging from software product lines to machine learning verification. Without uniformity, samples may be skewed toward certain regions of the solution space, leading to biased results and potentially overlooking critical configurations or solutions. Thus, maintaining uniformity guarantees that conclusions drawn from sampled data are both valid and generalizable.

Scalability refers to the efficiency of the sampler to produce samples within

a specified amount of time, even for large formulae. Achieving scalability while preserving uniformity is challenging because exact uniform sampling often relies on computationally expensive operations such as model counting or knowledge compilation, which can quickly become infeasible as formula size grows. Previous studies [PAP+19] demonstrated the difficulty for existing samplers to satisfy both quality criteria. Despite recent improvements [SGM20], state-of-the-art samplers still fail to scale on complex real-world formulae (representing, e.g., the Linux kernel configurations) without sacrificing uniformity. Therefore, scalability and uniformity are two conflicting objectives. True uniform random samplers with uniformity guarantees struggle to scale to larger formulae, and heuristic-based random samplers, which have better scalability, suffer from a lack of uniformity [PAP+19]. This trade-off forces practitioners to balance between the accuracy of sampling and the feasibility of running it on large instances.

A classic exact URS method is based on recursive Shannon decomposition and model counting. Suppose we have a formula F which is expressed over a set of variables $Var(F)$ with $a \in Var(F)$. Then we can sample from F by counting the total number of solutions to F, denoted by $|R_F|$, and by computing the number of solutions to $F \wedge a$, denoted by $|R_{F \wedge a}|$. We then know that to achieve uniformity we should set a to true with probability $P(a) = \frac{|R_{F \wedge a}|}{|R_F|}$. Once a value has been decided for a, we can sample from either $F \wedge a$ or $F \wedge \neg a$ depending on the value of a. Therefore, URS is closely related to model counting (#SAT) and knowledge compilation. This algorithm was implemented in Smarch [OGB+20]. Similar approaches, such as BDDSampler [HFG+22] and KUS [SGR+18], first compile the input formula into a different form (such as a Binary Decision Diagram (BDD)), and then run the recursive algorithm on the compiled form. Common compilation targets are BDDs and deterministic decomposable negation normal forms (d-DNNF), which allow for fast model counting. Therefore, compiling the formula to a different language avoids repeated (costly) calls to a model counter. An illustration of this advantage is given by Sharma et al. [SGR+18], who showed that sample size has little influence on the runtime of KUS.

Despite these advances, compilation-based samplers face limitations. Some formulae remain too large or complex to compile efficiently. To address this, heuristic-based samplers like CMSGen [GSC+21] and STS [EGS12] have been developed. These methods do not guarantee uniformity but scale better to large inputs. However, studies such as [PAP+19; HFG+22] show that heuristic methods often produce biased samples, compromising result quality. In summary, the landscape of URS methods remains defined by a fundamental tension: uniformity versus scalability.

1.2 Challenges

We have identified two main challenges in improving the scalability and reliability of URS.

1. **Lack of understanding of the average complexity.** #SAT is a well-known computational problem in the class #P, making it theoretically intractable in the worst case [Val79]. As a result, URS methods that rely on exact model counting inherit the same worst-case complexity. However, similar to SAT solving, practical tools such as sharpSAT [Thu06] and D4 [LM17] for #SAT, or SPUR [AHT18] and KUS [SGR+18] for URS, have been shown to be remarkably effective on many real-world formulae. Despite this empirical success, the reasons behind the performance of these tools remain poorly understood. In particular, it is unclear why certain industrial formulae are tractable for these methods while others, often of similar size or domain, are not. This lack of insight into average-case or practical complexity hampers progress in two ways: it limits our ability to predict whether a given formula is solvable with current techniques, and it makes it difficult to design targeted improvements for hard instances. Developing a deeper understanding of the empirical complexity landscape of URS and #SAT remains an open and important research challenge.

2. **Lack of computationally affordable uniformity tests.** At the time of writing, only two statistical tests exist for evaluating the uniformity of URS methods, most notably Barbarik [CM19] and SFpC [HFG+22]. While these tests are valuable, they are computationally expensive and often require significant time and resources to yield conclusive results. Additionally, Barbarik [CM19] requires a reference sampler which, if not uniform, can influence the final result. As a consequence, validating the uniformity of new URS approaches becomes a bottleneck, slowing down research and increasing development costs. This challenge is further exacerbated by the fact that formal proofs of uniformity may not always be feasible: either because the sampler is based on heuristics, or because the analysis required is too complex. Even when theoretical proofs exist, they typically apply to abstract models of the sampler, not to its actual implementation. In practice, implementations may diverge from the theoretical model due to bugs, optimisations, or unintended side effects — any of which can break uniformity. In such cases, empirical validation through statistical testing is the only way to detect violations of the theoretical guarantees. For example, the development of CMSGen [GSC+21] involved an iterative loop guided by Barbarik to refine the sampler and validate its behaviour. While effective, this approach is resource-intensive, highlighting the need for lightweight and robust statistical tests. Just as

statistical testing is used to validate the behaviour of pseudo-random number generators (PRNGs), efficient uniformity tests would serve as essential tools for both debugging and iterative development of URS methods. The lack of such tools remains a significant barrier to progress in the field.

1.3 Overview of Contributions

This dissertation brings four main contributions to address the above challenges and introduces a new divide-and-conquer approach to knowledge compilation with applications to URS.

- **Empirical study of the experimental complexity of URS and #SAT.** To address the first challenge, we conduct two extensive empirical studies on the average complexity of URS and #SAT. In the first study, we find that there is a correlation between the execution time (and memory) and various metrics such as the number of variables and the number of clauses. Additionally, we find that there is a strong correlation (Kendall coefficients > 60) with the minimal independent support (resp. the number of equivalence classes) of a formula and the execution time (and memory). We also demonstrate the positive role of *formula preprocessing*, i.e., simplifying the formulae before using URS or #SAT. Next, we lean on these results and develop a classification model to predict whether a given sampling problem (i.e., using a given uniform sampler to sample from a given formula) is affordable for a given time and memory budget. We evaluate our model on all our 488 subject formulae and show that it can achieve at best a classification F1-score of 0.97 and an AUC-ROC of 0.98.
- **Empirical study on phase transitions in URS and #SAT.** We extend our empirical investigation by studying phase transitions in the context of #SAT and URS. Phase transitions refer to abrupt changes in computational difficulty as certain structural parameters of a formula vary — a well-known phenomenon in classical SAT. In this contribution, we explore whether similar patterns emerge for #SAT and URS, with the goal of contributing to a more principled understanding of their average-case complexity. Our study combines two complementary approaches: empirical analysis of real-world formulae and controlled experiments on synthetic benchmarks. Real-world instances provide practical grounding, but their limited availability and high structural heterogeneity prevent systematic exploration of the full parameter space. To overcome this limitation, we design families of synthetic formulae using controlled generation procedures, allowing us to isolate and vary structural features such as constraint density and community structure. Our methodology enables us to identify complexity peaks and assess how formula structure influences scalability. Ultimately, the study deepens our understanding of

when and why URS and #SAT become computationally challenging, and lays the groundwork for future complexity predictions and algorithm selection strategies.

- **A set of five statistical tests for URS.** To address the second challenge, we develop a set of five statistical tests specifically designed for URS. Each test has different strengths and weaknesses and a different balance between its ability to detect non-uniformity and computational cost. We evaluate our statistical tests on a set of uniform random samplers and discover that most available samplers fail multiple tests except for UniGen3. Next, we explore another threat to the validity of uniformity testing: the input Boolean formula used for testing. Indeed, while one could consider uniformity as a universal property of sampling methods, the empirical nature of both sampler implementation and uniformity tests creates an inherent risk for a test to generate Type I and Type II errors. We, therefore, consider the use of multiple formulae for uniformity testing and present a methodology to combine results for individual tests into a statistically meaningful answer. Beyond this, we study the question of dataset bias. In particular, we investigate how test results obtained from synthetic formulae typically used in the URS research community correlate with results on real-world formulae extracted from SPL feature models. We find that while synthetic formulae can be designed to allow for much faster testing, further testing with real-world formulae remains necessary to obtain a high enough confidence in the uniformity result of the statistical tests. In other words, synthetic formulae can be used to quickly detect non-uniform samplers. However, some non-uniform samplers tend to pass the statistical tests if only synthetic formulae are used. Therefore, further testing with real-world formulae remains necessary.
- **A divide-and-conquer approach to knowledge compilation.** To improve the scalability of knowledge compilation, we propose DivKC, a divide-and-conquer strategy for compiling Boolean formulae into deterministic decomposable negation normal form (d-DNNF). The core idea is to decompose an input formula F into two smaller subformulae that can be compiled independently and more efficiently than F itself. This decomposition-based compilation has multiple benefits: it supports parallelisation and enables practical applications in both #SAT and URS. One key advantage of DivKC is that it yields, by construction, sound lower and upper bounds on the model count $|R_F|$, which can be refined using a statistical estimation method we introduce. This estimation procedure produces significantly tighter bounds than the initial decomposition provides, offering a practical approach for approximate model counting. As for URS, we can similarly simplify the resolution of this problem by successfully sampling from the two decomposed formulae.

2

Background

This chapter presents foundational concepts of Boolean formulae, model counting (#SAT), and uniform random sampling (URS), which will be used throughout this thesis.

Contents

2.1 Boolean Formulae

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

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. A formula is a k-CNF if every clause has exactly k distinct literals. For a CNF formula, we denote by $|F|$ its number of clauses.

An assignment a to the variables $Var(F)$ is a set of literals such that $\forall x \in a : (\neg x \notin a)$. We say that the literal l evaluates to true in a if $l \in a$, otherwise l evaluates to false in a. An assignment a is a partial assignment if $\exists x \in Var(F) : (x \notin a \wedge \neg x \notin a)$. We denote by $|a|$ the number of assigned variables. An assignment a is a complete assignment if $\forall x \in Var(F) : (x \in a \vee \neg x \in a)$. A model (or solution) 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 (or the model space 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 and 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 write $S \in {R_F}^N$ to denote a sample $S = (s_1, s_2, ..., s_N)$, an N-dimensional tuple containing models of F (i.e. $\forall i \in \{1, ..., N\} : (s_i \in R_F)$). By abuse of notation, we use $m \in S$ to denote that there exists $s_i = m$ for some $i \in \{1, ..., N\}$.

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 $\forall i \neq j : (R_{O_i \wedge O_j} = \emptyset)$.

We denote by $F|_a$ the conditioning of F with a (i.e., the propagation of the literals of a in F). 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 [Wan15; LLM16]. 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 $R_{Project(F,P)} = \{m \cap L | m \in R_F\}$, with $L = \{x, \neg x | x \in P\}$.

$I \subseteq Var(F)$ of a formula F is an *independent support* if every model of F can be uniquely distinguished by using the variables in I only [IMM+16; CMV14] (i.e., $|R_{Project(F,I)}| = |R_F|$). An independent support is minimal (MIS) if removing any

variable from it does not yield an independent support.

Based on the above, we define the concepts of backbone and *equivalence class*:

Definition 1 (Backbone)**.** *The backbone B_F of a formula F is defined as the set of literals that appear in each model of the formula: $B_F = \bigcap_{m \in R_F} m$.*

The backbone contains the literals that evaluate to true in every model of the formula. If we generalise the idea of backbone, we find the notion of equivalence class:

Definition 2 (Equivalence class)**.** *An equivalence class e is a set of literals that evaluate to the same value in every model of F.*

$$\forall l, l' \in e : \forall m \in R_F : ((l \in m) \Leftrightarrow (l' \in m))$$

By this definition, we find that if $\{x, y\}$ is an equivalence class, then $\{\neg x, \neg y\}$ is also an equivalence class. These two equivalence classes are redundant as they represent the same result. We define two equivalence classes a and b as redundant if and only if $a = b$ or $a = \{\neg x | x \in b\}$ or $a \subseteq b$ or $b \subseteq a$. If we have $a \subseteq b$, we keep b and discard a. For the rest of the paper, without loss of generality, we only consider non-redundant equivalence classes. We define the set E_F as the set of all non-redundant equivalence classes of a formula F and $|E_F|$ as the number of equivalence classes of F. Note that we necessarily have $|E_F| \leq |Var(F)|$ because we only consider non-redundant equivalence classes.

We next define three common problems for Boolean formulae: SAT solving, model counting, and URS.

Definition 3 (SAT solving)**.** *SAT solving is the problem of determining whether R_F is non-empty.*

Definition 4 (Model Count)**.** *Model Counting (# SAT) is the problem of computing $|R_F|$.*

Definition 5 (Uniform Random Sampling)**.** *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.*

2.2 Community Structure of Boolean Formulae

An undirected weighted graph G is defined as a pair $G = (V, w)$, where V is the set of nodes and w is the edge-weight function defined as $w : V \times V \longrightarrow \mathbb{R}^+$. Because G is undirected, we have $w(x, y) = w(y, x)$.

The Variable Incidence Graph (VIG) [GL15] of a CNF formula F is the undirected weighted graph whose nodes are the variables of F. There exists an edge between

two variables if they both appear in a clause c. To give the same relevance to all clauses, we define the weight of an edge between nodes x and y as $w(x,y) = \sum_{c \in F, x \in Var(c) \wedge y \in Var(c)} \frac{1}{\binom{|c|}{2}}$, with $|c|$ the number of literals in the clause.

A formula has a community structure if we can split the variables into at least two groups such that we have a higher number of clauses that connect variables within a group than clauses that connect multiple groups. This is an interesting property for model counting and sampling because if we have a formula that does not have any connection between the groups (i.e., no clauses connecting groups), then we can compute the model count (or sample) of each group separately and compute the product to obtain the final result. Thus, if we have a high community structure, we expect that the algorithm will finish faster.

To measure the community structure of a formula F, we will use the notion of modularity Q as defined in [NG03; GL15; AGL12] computed on the VIG G of the formula F.

The modularity of a graph G is defined for a given partition C as follows:

$$Q(G,C) = \sum_{C_i \in C} \frac{\sum_{x,y \in C_i} w(x,y)}{\sum_{x,y \in V} w(x,y)} - \left(\frac{\sum_{x \in C_i} deg(x)}{\sum_{x \in V} deg(x)} \right)^2$$

The modularity of a graph is $Q(G) = max\{Q(G,C)|C\}$ for any partition C. Computing the modularity of a graph is NP-hard [BDG+08], thus most methods usually approximate a lower-bound of Q. The modularity of a graph will be in the range $[0,1]$ [AGL12], with one meaning a very strong community structure, and zero meaning that the graph is fully connected.

3

Related Work

This chapter reviews related work on uniform random sampling and model counting.

Contents

3.1 Model Counting

Model counting can be performed using specialised algorithms such as sharpSAT [Thu06], GANAK [SRS+19], and McTW [FHH20], which directly compute the number of satisfying assignments to a Boolean formula. An alternative strategy is knowledge compilation, where the formula is transformed into a structured representation, such as a d-DNNF, that enables efficient and repeated model counting. This compilation-based approach not only supports exact inference but also facilitates reuse across multiple queries, as demonstrated by Sundermann et al. [SRH+24], who also highlighted its practical limitations [SHN+23]. Several knowledge compilers exist, including C2D [Dar+04], D4 [LM17], and DSharp [MMB+12], which can handle large formulae. However, many formulae remain beyond the reach of current compilers. This is not surprising, given that model counting is #P-complete, as established by Valiant [Val79], and thus remains computationally intractable in the general case.

To make better use of the available computing resources, Lagniez et al. [LMS18] proposed DMC, a distributed model counter. 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.

Approximate model counters have been proposed to overcome the lack of scalability of exact model counting. A notable approximate model counter is ApproxMC [YM23; PMY25], which provides theoretical guarantees on the quality of the model counts. ApproxMC is a hashing-based algorithm to compute approximate model counts with strong theoretical guarantees. These strong theoretical properties come at a cost: the hashing-based approach requires adding large clauses to formulae so they can be sampled. These clauses grow quadratically in size with the number of variables in the formula, which can raise scalability issues. Other approximate model counters are ApproxCount [WS05], which offers no guarantees, and SampleCount [GHS+07], which returns a lower bound to the true model count with high confidence. However, current approximate model counters have 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.

3.2 Uniform Random Sampling

Uniform random sampling is a problem related to model counting. Model counting can be used to sample uniformly at random from a formula, as is done by SPUR [AHT18] and Smarch [OGB+20]. Smarch [OGB+20] produces models by recursively assigning values to each variable of the formula. At each step, a

model counter is called with the formula and the current partial assignment to compute the probability distribution of the next variable. SPUR [AHT18] is tightly integrated with sharpSAT [Thu06] and is based on reservoir sampling [Vit85]. Reservoir sampling is a family of algorithms that operate on a stream. The models of the formula are streamed into the algorithm, and with each model, the algorithm updates an internal reservoir. Once all the models have been streamed, a uniform random sample can be extracted from the reservoir.

A similar approach to uniform random sampling is based on knowledge compilation. The formula is first compiled to BDD or d-DNNF, and the compiled form is then used to produce samples. Notable examples are BDDSampler [HFG+22], which is based on BDD, and KUS [SGR+18], which is based on d-DNNF.

The samplers based on model counting and knowledge compilation provide theoretical guarantees of uniformity. Another approach to theoretically guaranteed uniformity is UniGen3 [SGM20]. UniGen3 [SGM20] is a hashing-based algorithm that generates models in a nearly uniform manner with strong theoretical guarantees: it either produces models satisfying a nearly uniform distribution or it produces no model at all. Similarly to ApproxMC, these strong theoretical properties come at a cost: the hashing-based approach requires adding large clauses to the formula.

To overcome the scalability challenges of true uniform random samplers, several heuristic-based alternatives have been proposed, including QuickSampler [DLB+18], STS [EGS12], and CMSGen [GSC+21]. While these methods lack formal guarantees of uniformity, they are significantly more scalable in practice [PAP+19].

QuickSampler is built on a set of heuristics designed to quickly generate assignments across a wide range of industrial benchmarks [DLB+18]. However, it offers no guarantees regarding the distribution, validity, or even termination of the sampling process, therefore requiring post-verification using a SAT solver.

STS, in contrast, is a SAT-solver-based approach that recursively constructs valid partial assignments, pruning invalid ones at each step. When the number of partial assignments grows beyond a threshold, a random sub-sample is selected. This process continues until complete assignments (models) are obtained. While STS uses a SAT solver to ensure validity, it only approaches uniformity when the parameters are large enough to enable near-complete model enumeration.

CMSGen [GSC+21] adopts a different strategy: its authors used the uniformity testing tool Barbarik [CM19] to tune the parameters of the SAT solver CryptoMiniSAT [SNC09], with the goal of empirically achieving uniform sampling.

While heuristic-based samplers are interesting, especially when knowledge compilation-based sampling does not scale, they offer no uniformity guarantees, and sampling quality can vary depending on the formula [PAP+19].

3.3 Empirical Sudies on Complexity

As noted by Alyahya et al. [AMM22] and Ganesh and Vardi [GV21], studying the complexity of SAT-based tasks is not new. One of the first approaches was to characterise *phase transitions* linked to abrupt changes in solving complexity. Monasson et al. [MZK+99] offered a structural metric, namely the clause-to-variable ratio. They demonstrated that when this ratio increases, finding models for a given synthetic formula is progressively harder up to a critical value of this ratio. When the ratio exceeds this critical value, the formula becomes easy to solve again (often by proving it UNSAT). Alyahya's survey further covers metrics such as treewidth correlated with solving time [Mat11].

Regarding FM-based formulae specifically, the body of knowledge is more limited. Mendonca et al. [MWC09] studied the experimental complexity of SAT-solving for FM-based formulae. The authors studied the clause-to-variable spectrum of formulae similar to feature models. In their studies, the authors failed to observe a phase transition and concluded that FM formulae do not suffer from the SAT phase transition, thus explaining the general efficiency of SAT-based analysis of feature models. Liang et al. [LGC+15] further confirmed these results on larger industrial FMs. The authors found that FMs have a high number of unrestricted variables due to the high variability of FMs. The authors also found that SAT-solvers do little backtracking during search, thus explaining the high efficiency. The authors followed by disabling SAT solver heuristics and found that the solver did not suffer from any performance deterioration while solving FMs. In addition to this extensive analysis, the authors ran a set of simplifications to the formulae and found that they were highly efficient. Some of the instances were solved by the simplification procedure alone. The remaining formulae were small in comparison and thus efficiently solved by state-of-the-art SAT-solvers. Johansen discussed the implications of these findings for combinatorial interaction testing of software product lines [JHF11].

The body of literature on #SAT and uniform random sampling remains relatively sparse. Sundermann et al. [SHN+23] evaluated 21 #SAT solvers on FM-based formulae and analysed correlations between solver runtime and formula metrics. Plazar et al. [PAP+19] investigated the scalability of the samplers UniGen [CMV14] and QuickSampler [DLB+18], although the analysis is limited to UniGen, as QuickSampler does not provide theoretical guarantees of uniformity. Escamocher and O'Sullivan [EO22] explored the generation of hard instances for #SAT. Several works examined phase transitions in related domains: [HD04; BL99; BP00] observed phase transitions in knowledge compilation; Gupta et al. [GRM20] extended this by analysing phase transitions in d-DNNF, SDD, and OBDD representations; and Gao et al. [GYX11] conducted a similar study. However, these works do not address phase transitions in URS, nor do they investigate the role of community structure or

solution density, as proposed by Gupta et al. [GRM20]. Overall, phase transitions in URS and the impact of structural properties like community structure on solving times remain unexplored.

3.4 Uniformity Testing

Assessing the quality of a pseudo-random number generator (PRNG) is similar to assessing the quality of a uniform random sampler. One key property desired by both is that every possible value should have a uniform probability of being returned. In the case of URS, this means returning a model to a Boolean formula at random. In the case of PRNGs, this means returning an integer (on a usually predefined number of bits) uniformly at random. In other words, a PRNG is a uniform random sampler for a Boolean formula with no constraints. Statistical testing of PRNGs to assess their quality is quite common. One test suite is the NIST test suite developed by Rukhin et al. [RSN+01]. Other, more recent test suits include TestU01 [LS07], dieharder [BEB18] and PractRand [Dot]. All of these test suites contain multiple statistical tests. Each test has different strengths and weaknesses. Thus, each test is suitable to detect different kinds of weaknesses in the generated stream of numbers. For example, the monobit test checks whether the number of ones and zeroes is approximately equal in the binary stream of numbers generated by a PRNG. Therefore, a PRNG passes the monobit test if it generates a stream of alternating ones and zeros. However, the PRNG would produce 'random' numbers of poor quality, thus highlighting the need for multiple tests.

Unfortunately, a large number of tests performed on PRNGs rely on the fact that the solution hyperspace is unconstrained. Moreover, PRNGs are much faster at generating samples than uniform random samplers. This is shown by Blackman and Vigna [BV21] who tested PRNGs on up to 10^{15} generated bytes. This is feasible as PRNGs are engineered to generate 32-bit or 64-bit words in the nanosecond or even sub-nanosecond range. However, URS implementations often rely on NP-Oracles such as SAT-solvers [EGS12] or on #P-Oracles [AHT18; SGR+18; Val79]. This implies that URS is much slower than pseudo-random number generation. Thus, URS needs its own set of specialised tests that can work with small sample sizes and under the constrained hyperspace implied by the Boolean formula given as input to the sampler under test.

As indicated by Plazar et al. [PAP+19], assessing the uniformity distribution of SAT solutions is difficult, and a direct method is prohibitively expensive. Barbarik by Soos et al. [SGM20] is a test which, by using a uniform random sampler as reference, tests whether a given input sampler is uniform or not. In other words, Barbarik tests whether both samplers sample from the same distribution. If the reference sampler is uniform, then Barbarik tests the uniformity of the sampler under test. The authors updated Barbarik [SGC+22] to support a more fine-grained

analysis of uniformity. The approach has the main downside of requiring a uniform random sampler as a reference. If the reference sampler is not uniform, then the results are unreliable. A high level of trust is thus required for the reference sampler. Another way of assessing the solutions' uniformity is the statistical test proposed by Heradio et al. [HFG+22]. While the two kinds of approaches seem to agree on the (non-)uniformity of most samplers, they seem to disagree on Smarch's status [OGB+20]. It is currently an open question of whether this disagreement stems from a different test design or the selection of SAT formulae, which, though overlapping, are not exactly the same. We note that there is a close relationship between the design of URS techniques and testing uniformity: the improvements of Barbarik led to CMSGen [GSC+21], while Heradio et al. [HFG+22]'s uniformity tests led them to develop BDDSampler, a novel uniform sampler based on binary decision diagrams.

4

Preprocessing is What You Need: Understanding and Predicting the Complexity of SAT-based Uniform Random Sampling

Understanding why a formula is difficult to sample from is a challenging problem. Multiple factors, along with their interactions, can affect the formula's hardness. In this chapter, we examine several structural metrics and investigate potential methods to predict the computational cost of URS and #SAT.

Contents

4.1 Introduction

Why some formulae are harder to sample uniformly is *a poorly understood problem*. A simple but wrong approach to determining sampling complexity is to count the number of variables and clauses of formulae. As an example, UniGen3 [SGM20] requires 8 seconds to produce 10000 models from the formula `blasted_case64` — 96 variables and 299 clauses — and 13.5 seconds for the same number of models from the `JHipster` feature model — 44 variables and 104 clauses. This indicates that these simple metrics do not adequately characterise the complexity of sampling. While there exist formula metrics that correlate with the complexity of SAT *solving* — although with varying successes [AMM22] — the characteristics that make a formula easier or harder to sample from remain unknown.

In this chapter, we assess and define meaningful metrics for understanding and predicting URS difficulty (time and memory consumption). In addition to simple metrics trivially computed from the formula structure, we consider other studied metrics in the context of SAT solving (such as the minimal independent support size and the treewidth). We also provide an efficient algorithm to compute equivalence classes [CFM13]. To evaluate the relevance of these metrics to assess sampling and solving complexity, we consider two uniform random samplers — SPUR [AHT18] and UniGen3 [SGM20] — as well as SAT solvers [DB08; ES03] and a model counter [LM17]. Motivated by previous studies showing that the formulae encoding the variability spaces of configurable systems tend to be harder to uniformly sample than others [PAP+19; GSC+21], we evaluate our metrics on a diversified dataset (made available by Plazar et al. [PAP+19]) of 488 SAT formulae, 128 of which encode configurable systems.

Equipped with a set of metrics measured on various formulae, we measure correlations between these metrics and the performance of uniform samplers. We demonstrate the existence of strong correlations (Kendall coefficients > 60) between some (combinations of) metrics and sampling complexity (time and memory consumption). We also demonstrate the positive role of *formula preprocessing*, i.e., simplifying the formulae and applying the metrics on the preprocessed formulae, for complexity prediction. Next, we lean on these results and develop a classification model to predict whether a given sampling problem (i.e., using a given uniform sampler to sample from a given formula) is affordable for a given time and memory budget. We evaluate our model on all our 488 subject formulae and show that it can achieve at best a classification F1-score of 0.97 and an AUC-ROC of 0.98.

To summarise, this chapter makes the following contributions:

1. ***Correlation study.*** We study the correlation between the complexity metrics and the computational cost of sampling (time and memory). We demonstrate a strong correlation between the number of equivalence classes and sampling cost.

2. ***Prediction.*** Based on these correlations, we build classification models (random forests) that leverage the metrics to classify formulae according to sampling cost, with F1-score up to 0.97 and AUC-ROC up to 0.98. We further analyse the feature importance of these models to increase our trust in the correlation study.

4.2 Objectives and Methods

Our objective is to understand and predict the capability (or lack thereof) of state-of-the-art samplers to sample models from a Boolean formula uniformly.

4.2.1 Research Questions

Our first research question investigates the role of metrics in the complexity of uniform random sampling:

RQ1: Which metrics of Boolean formulae correlate with URS time and memory consumption?

In addition to simple characteristics like the number of variables and clauses, we consider concepts that are intensively used in the problems of SAT solving, model counting, and URS, e.g., the size of the minimal independent support and the number of *equivalence classes*.

We aim to exploit our analysis results to develop an approach that, based on the correlated characteristics, can predict whether a formula would be too costly to uniformly sample from (i.e., would exceed a predefined time and memory budget). This would enable engineers to estimate whether it is feasible to sample models with uniform samplers *without* wasting computation resources on intractable problems.

RQ2: Can the correlated characteristics be used to predict the affordability of URS in terms of time and memory consumption?

To answer this question, we train random forest models to classify Boolean formulae into 'affordable' or 'not affordable', based on different combinations of the characteristics we study.

Lastly, we study whether the intrinsic links between SAT solving, model counting, and sampling translate into the same influence of formula characteristics on these three problems.

RQ3: Are the characteristics of Boolean formulae correlated to the complexity of URS, as they are to SAT solving and model counting?

A positive answer to this question would pave the way to improve the efficiency of URS by working on the same formula transformations that reduce the difficulty of SAT solving and model counting. A negative answer would invalidate this path and call for specific solutions to reduce the complexity characteristics that impact sampling.

4.2.2 Complexity Metrics

We consider simple metrics that are trivially computed from the structure of a Boolean formula:

- the number of variables $|Var(F)|$
- the number of clauses $|F|$
- the number of literals $\#l$

We, furthermore, consider underlying concepts that SAT solvers, counters, and samplers have used to improve the performance of their algorithms. One such metric is $\#mis$, *the size of the Minimal Independent Support* (MIS). MIS is typically computed to improve the performance of model counters (like D4 [LM17] and sharpSAT [Thu06]) that some URS tools invoke during sampling.

Unfortunately, computing the MIS itself may be unaffordable for complex formulae. To this end, we propose the number of *equivalence classes* ($|E_F|$). The advantage over MIS is that the computation of equivalence classes only requires a simple SAT solver. We further increase the efficiency of this computation through a parallel algorithm that we develop hereafter. Using this algorithm, we compute $|E_F|$ for the Linux 2013 model (50000 variables) [PTR+19] in less than 1.5 wall-clock hours while computing $\#mis$ times out at 24 hours. Another way of approximating the MIS is to use Arjun [SM21], which is significantly faster than the computation of *equivalence classes*. Unfortunately, using Arjun to compute an independent support gave us lower correlations; therefore, we decided to use MIS [IMM+16] and $|E_F|$. In addition, we consider other metrics that have been studied in the context of SAT solving, viz. treewidth [OD14] and deficiency [PS19]. Treewidth (tw) is used to bound the worst-case size of the decision DNNF (D-DNNF) during solving [OD14]. Deficiency (δ) was proven to have intrinsic links with the worst-case time complexity of SAT solving [PS19]. Though computing deficiency is an NP-hard problem, it can often be approximated as the number of clauses minus the number of variables.

4.2.3 EQV: A Parallel Algorithm to Compute the Number of Equivalence Classes

In [CFM13], the authors generalise the notion of backbone to equivalence classes and propose an algorithm to compute the equivalence classes. However, their algorithm requires adding $\frac{n(n-1)}{2}$ variables to a formula with n variables — in our dataset, n can be as high as 486193 variables. Assuming every variable requires

4 bytes of RAM to be stored, the algorithm would necessitate around 472 GB of RAM to store the additional variables. This is unaffordable and prevents us from computing $\#eqv$ on most of the formulae we use in our experiments.

We therefore propose an adapted algorithm that requires less storage memory and can improve efficiency via parallelisation. Our algorithm can divide the computation of [CFM13] to reduce the number of added variables and enable spreading over multiple cores. It introduces an overhead, though, as it may increase the number of intermediate solver calls. As a result, our approach would run slower on a single-core computer than [CFM13], but brings benefits on multi-core infrastructures.

Our method is depicted by Algorithm 1 with $\oplus$ being the logical exclusive or operator. The algorithm uses a *SAT* procedure, which takes as input a Boolean formula and either returns *UNSAT* if the formula is not satisfiable or returns the set of literals that represents the model found by the SAT solver.

The algorithm begins with an initial call to the SAT solver, which returns a model m, under the assumption that the formula is satisfiable. The algorithm then initializes two partitions of m, namely e and v. We use e to track separations — i.e., literals that are not in the same equivalence class, and v to track unions — i.e., equivalences for which we have proof that they hold. Thus, the partition e represents the set of possible but unverified equivalence classes.

From the candidates in e (lines 6–9), the algorithm repeatedly selects a pair $\{x, y\}$ that are considered equivalent in e but not yet in v. We then assume that x and y are equivalent and attempt to prove this assumption. To do so, we call the SAT solver and request a model in which x and y are different (i.e., $SAT(\phi \wedge (x \not\Leftrightarrow y))$), aiming to disprove their equivalence.

If the solver returns *UNSAT*, the equivalence is proven, and v is updated by merging the sets that contain either x or y. On the other hand, if the SAT solver returns a model *tmp*, we know that there exists a model of the formula in which x and y are not equivalent, and therefore they cannot be in the same equivalence class. We can also learn from the model *tmp* by comparing it with the initial model m. If two literals x and y are supposed to be equal in all models, then the presence of x in both m and *tmp* should imply the presence of y as well. In other words, any change in x from m to *tmp* should be mirrored by a change in y. Using this insight, we update e between lines 18 – 24 by further splitting its elements.

Therefore, v contains the verified equivalence classes, and e captures confirmed separations. The partition e thus helps avoid unnecessary *SAT* calls when two models have already shown that two literals are not equivalent.

The loop on line 6 is the for loop that may be parallelised. The critical sections of Algorithm 1 may seem very large, but the data structures e and v can be updated efficiently (especially considering that v may be implemented using the UnionFind data structure). Moreover, the *SAT* calls are done outside of a critical section and

Algorithm 1 EQV(ϕ)

Require: ϕ a satisfiable Boolean formula

1: $m \leftarrow SAT(\phi)$
2: $e \leftarrow \{m\}$
3: $v \leftarrow \{\{x\} | x \in m\}$
4: $crit \leftarrow mutex$
5: **do in parallel**
6: **for all** $\{x, y\} \in \mathcal{P}(m)$ **do**
7: $lock(crit)$
8: $C \leftarrow (\exists i \in e : \{x, y\} \subseteq i) \wedge \neg(\exists i \in v : \{x, y\} \subseteq i)$
9: $release(crit)$
10: **if** C **then**
11: $tmp \leftarrow SAT(\phi \wedge (x \not\Leftrightarrow y))$
12: $lock(crit)$
13: **if** $tmp = UNSAT$ **then**
14: {the SAT solver proved the equivalence of x and y}
15: $t \leftarrow \bigcup_{i \in v | x \in i \vee y \in i} i$
16: $v \leftarrow \{i | i \in v \wedge x \notin i \wedge y \notin i\}$
17: $v \leftarrow v \cup \{t\}$
18: **else**
19: {the SAT solver disproved the equivalence of x and y}
20: $r \leftarrow \emptyset$
21: **for all** $i \in e$ **do**
22: $a \leftarrow \{l | l \in i \wedge l \in tmp\}$
23: $b \leftarrow \{l | l \in i \wedge \neg l \in tmp\}$
24: $r \leftarrow r \cup \{a\} \cup \{b\}$
25: **end for**
26: $e \leftarrow r$
27: **end if**
28: $release(crit)$
29: **end if**
30: **end for**
31: **return** v

thus in parallel, which should grant us a significant speedup.

4.3 Experimental Setup

We detail below the general experimental protocol that applies to all research questions. The specific settings of each research question are detailed in Section 4.4.

4.3.1 Samplers

For this study, we use both SPUR and UniGen3 as these are the state-of-the-art samplers with theoretical guarantees of uniformity.

In our study, we would also like to explore the relationship between URS and SAT solving and the relationship between URS and SAT counting. To compare URS with SAT solving, we explored the two solvers MiniSAT [ES03] and Z3 [DB08]. To compare with SAT counting, we used the two state-of-the-art model counters D4 [LM17] and sharpSAT [Thu06]. Since another sampler called KUS [SGR+18] is based on D4, this should also give us insights into the complexity of KUS. We do not evaluate KUS as most of the complexity related to the sampling process is absorbed by the call to D4 as demonstrated in [SGR+18].

We added an implementation of bounded SAT solving (BSAT) using Z3. BSAT is a function BSAT(ϕ, n) defined as follows: the function recursively calls Z3 on ϕ and removes the returned model from the formula until either the formula becomes unsatisfiable or the number of iterations is greater than n. BSAT is thus a form of SAT sampler which is almost guaranteed to be very far from uniform.

4.3.2 #SAT Preprocessing

We would like to study the influence of formula preprocessing on the complexity of URS and the correlations with our metrics. To this end, we use a preprocessor called Arjun [SM21]. Arjun computes an independent support I of the input formula F and removes the variables that are not in the independent support I if the projection can be done in a reasonable time and space. We thus obtain a new formula F', which is the projection of F on the set of variables I. Arjun ensures that $R_{F'}$ is the projection of R_F on the independent support I and that $|R_{F'}| = |R_F|$. Thus, using Arjun as a preprocessor to URS does not influence the uniformity of a sampler if the sampler is guaranteed to be uniform.

4.3.3 Dataset

We use well-known and publicly available models in our study, which are of various complexities and are either feature models or general Boolean formulae.

Feature Model Benchmark.

Overall, we use the feature models of 128 real-world configurable systems (Linux, eCos, toybox, JHipster, etc.) with varying sizes and complexity. We first rely on 117

feature models used in [KTM+17; KTS+18]. The majority of feature models contain between 1,221 and 1,266 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295, and the other nine have between 49,770 and 50,606 cross-tree constraints [KTM+17; KTS+18]. Second, we include ten additional feature models used in [LGC+15] and not in [KTM+17; KTS+18]; they also contain a large number of features (e.g., more than 6,000). Third, we add the JHipster feature model [Rai15; HNA+18] to the study, a realistic but relatively small feature model (45 variables, 26,000+ configurations). We later refer to these benchmarks as the feature model benchmarks. Once put in conjunctive normal form, these instances typically contain between 1 and 15 thousand variables and up to 340 thousand clauses. The hardest of them, modelling the Linux kernel configuration, has more than 6,000 variables and 340,000 clauses. It is generally seen as a milestone in configurable system analysis.

General Boolean Formulae.

In addition to these feature models, we have replicated the initial experiments on industrial SAT formulae as conducted in [DLB+18]. We use these results to ensure that we are using the tools with the same configurations that were previously compared. Moreover, since these original formulae are much smaller than the feature models we use (typically a few thousand clauses), they will provide a basis for statistical analysis in case a solver cannot produce enough samples on the harder formulae.

Both of these datasets, the feature model benchmark and the general Boolean formulae, have been collected by Plazar et al. [PAP+19].

4.3.4 Infrastructure

The experiments regarding the computation of the equivalence classes, the MIS computation, as well as the time and memory usage of the samplers were computed on an HPC containing 318 nodes, each of which has 256 GB of RAM and 2 AMD Epyc ROME 7H12 CPUs running at 2.6 GHz.

To measure the memory usage of the samplers, we developed a wrapper program that reads the appropriate file in the /proc folder, which contains information about the virtual memory usage of the program. We asked the samplers to compute 1000 models while using less than 64 GB of RAM and in under 5 hours.

The treewidth was computed with the tool described in [HS15]. The correlations were computed using the SciPy Python library. To train the predictors, we used Python and the scikit-learn library [PVG+11]. We used standard parameters for random forests. We set the number of trees to 100, used Gini impurity for splitting, and set the number of features to consider at each split to the square root of the total number of features.

4.4 Results

4.4.1 RQ1: Complexity Factors

	$\lvert Var(F)\rvert$	$\lvert F\rvert$	#l	tw	δ	#mis	$\lvert E_F\rvert$	Z3 time	Z3 mem
SPUR time	45.7	48.5	50.0	34.5	49.2	**62.2**	**68.4**	34.2	39.9
SPUR mem	42.2	46.1	47.9	31.5	47.0	**60.1**	**62.7**	32.6	37.9
SPUR (+Arjun) time	58.8	**79.1**	**75.6**	**64.0**	**75.5**	50.8	-	34.5	46.5
SPUR (+Arjun) mem	**61.9**	**79.1**	**75.2**	**64.1**	**74.2**	53.9	-	35.6	47.5
UniGen3 time	47.2	45.5	45.0	34.6	44.1	54.5	**74.8**	25.0	22.2
UniGen3 mem	47.8	45.2	45.0	37.6	43.4	**68.6**	**71.3**	24.8	24.4
UniGen3 (+Arjun) time	**88.9**	46.0	44.7	41.6	44.0	**81.1**	-	21.9	32.5
UniGen3 (+Arjun) mem	**86.9**	38.8	37.9	35.2	36.5	**88.1**	-	19.4	29.5

Table 4.1: Kendall rank correlation coefficients of the used metrics with SPUR (416 data points), SPUR (+Arjun) (441 data points), UniGen3 (241 data points) and UniGen3 (+Arjun) (309 data points). All of the p-values are lower than 0.001.

Table 4.1 shows the Kendall rank correlation coefficients for the SPUR and UniGen3 samplers. The coefficients have been computed on the instances on which we successfully managed to compute 1000 models in less than 5 hours, and using less than 64GB of virtual memory. This means that the table was computed on 416 formulae for SPUR and 241 formulae for UniGen3. The columns $|Var(F)|$, $|F|$, and #l represent, respectively, the number of variables, the number of clauses, and the number of literals, with the number of literals being the sum of the lengths of all clauses. The time and mem columns indicate the computation time and the amount of virtual memory used by a single call to Z3, respectively. We have two groups in our table, the regular group where we compute the correlations over our formulae, and the (+Arjun) group where we first preprocess the formula with Arjun [SM21] and then call SPUR or UniGen3 on the output of Arjun. Some solvers take advantage of a possible MIS declaration inside the DIMACS files. Unfortunately, not all of the solvers take advantage of the MIS declaration. We thus removed the MIS declarations from the DIMACS files. The results with the MIS declaration are nonetheless available on our companion GitHub [Zey23]. There are no correlations between the (+Arjun) groups and the equivalence classes because Arjun automatically removes redundant variables. The time and memory usage of Arjun is ignored (the median runtime was 0.15 seconds, with the longest runtime being 17 minutes). All the p-values are lower than 10^{-3}. We computed the MIS by using the tool in [IMM+16] on both the initial formulae and the preprocessed formulae. Although Arjun [SM21] returns an independent support, we find that the correlations are worse. We thus decided to compute the MIS with [IMM+16].

For both SPUR and UniGen3, we observe that the most correlated metrics with the computation time or the virtual memory usage are either the size of the MIS

or the number of equivalence classes. However, if we add Arjun as a preprocessing step, we observe that the correlations change between SPUR and UniGen3. SPUR (+Arjun) is highly correlated with the number of clauses and with δ, while UniGen3 (+Arjun) is highly correlated with the number of variables and the size of the MIS. This difference can be explained through their respective algorithms. UniGen3 adds clauses to the formula, and the size and number of added clauses depend on the number of variables (or on the MIS if the MIS is declared in the DIMACS file). SPUR, on the other hand, is based on an exhaustive DPLL algorithm, which means that SPUR spends a lot of time doing Boolean constraint propagation, which is sensitive to the number of clauses.

Answer to RQ1: The number of equivalence classes and the number of variables in the MIS strongly correlate (Kendall rank correlation coefficients > 62 for all formulae) with computation time and memory usage of both UniGen3 and SPUR. If the formulae are preprocessed with Arjun, then we find that the highest correlations are with the number of variables, the number of clauses, δ, and the size of the MIS.

4.4.2 RQ2: Complexity Prediction

We cover here the results regarding formula classification using our trained random forests. We consider binary classification here. We selected the formula processed within the following affordability limits: 30 minutes of computation time and less than 4GB of virtual memory. This selection allowed balanced training data.

	$\lvert Var(F)\rvert$	$\lvert F\rvert$	#l	tw	δ	#mis	$\lvert E_F\rvert$	Z3 time	Z3 mem
SPUR	9.8	6.5	4.6	10.9	7.7	13.0	**37.8**	3.3	5.9
UniGen3	7.0	8.3	11.0	2.7	8.7	**27.0**	**31.6**	1.4	1.9
SPUR (+Arjun)	4.0	16.2	**20.8**	**25.3**	17.8	2.9	-	2.9	9.8
UniGen3 (+Arjun)	**27.6**	**20.9**	12.6	4.4	10.5	18.2	-	1.8	3.5

Table 4.2: Feature importances in a random forest containing 1000 instances

Table 4.2 shows the different Gini importances (i.e., feature importances) of our different metrics in a random forest that contains 1000 instances. The lines where the SAT sampler is suffixed with '(+Arjun)' are the lines where the formulae were first preprocessed with Arjun [SM21]. The time and memory used for a single Z3 call play a negligible role. The two main features are the number of equivalence classes and the size of the MIS. If, however, we use Arjun as a preprocessor, we observe that the number of variables, the number of clauses, the number of literals, and δ seem to be interesting choices as well, further confirming our initial correlations. The

treewidth has high importance for SPUR (+Arjun) but is expensive to compute, diminishing its value for large formulae.

	SPUR	SPUR (+Arjun)	UniGen3	UniGen3 (+Arjun)
#mis	67.4	73.3	91.3	90.9
$\|E_F\|$	80.9	-	91.2	-
δ'	62.7	85.7	83.0	90.7
$\|Var(F)\|$	60.0	62.4	86.6	93.7
$\|F\|$	65.4	86.1	85.4	91.4
all	83.9	**91.2**	97.3	96.7
$\|Var(F)\|$, $\|F\|$, #l, tw, δ, #mis, $\|E_F\|$	**85.1**	**92.6**	**97.9**	**96.9**
$\|Var(F)\|$, $\|F\|$, tw, δ, #mis, $\|E_F\|$	83.2	91.0	**98.0**	96.7
$\|Var(F)\|$, tw, δ, #mis, $\|E_F\|$	84.4	91.0	97.8	**96.9**
$\|Var(F)\|$, tw, δ', #mis, $\|E_F\|$	83.0	**92.6**	**98.4**	**96.9**
$\|Var(F)\|$, δ', #mis, $\|E_F\|$	**83.5**	-	97.8	-
$\|Var(F)\| + \delta' +$ #mis	78.7	89.2	97.1	96.2
$\|Var(F)\| + \delta' + \|E_F\|$	**83.3**	-	94.6	-
$\|Var(F)\| + \delta'$	68.6	88.5	89.5	95.8
$\|Var(F)\| + \|F\| + \delta'$	66.6	90.1	89.5	95.8
$\|Var(F)\| + \|F\| +$ #l $+ \delta'$	68.7	88.7	89.8	95.8

Table 4.3: F1-scores with different features of a random forest containing 100 instances estimated using LOO

In Table 4.3, we explore the F1-scores of a random forest containing 100 instances that were trained on different metrics. The 'all' line indicates the predictor trained on all of the metrics. We also use δ' instead of δ in some of the experiments. δ' is defined as $\delta' = |F| - |Var(F)|$. While this is only an estimation of δ, our experiments show that it is usually a very good estimation, and it is a lot faster to compute as well. As previously reported, we report sampler results with and without the Arjun preprocessing step. $|E_F|$ is always ignored when Arjun is used as a preprocessor. Arjun automatically simplifies the equivalence classes in the formula; thus, we find $|Var(F)| = |E_F|$ for the preprocessed formulae, eliminating the need to compute $|E_F|$. The table entries that involve both Arjun and $|E_F|$ are simply computed by ignoring $|E_F|$. The predictions were done using a leave-one-out strategy, and the F1-scores were evaluated on the predictions. This means that for every data-point x, we trained a model on the complete data-set excluding x and performed a prediction for x. The predictions are collected in a table, and the scores are computed on the prediction table. Table 4.4 shows the ROC AUCs, just like Table 4.3 shows the F1-scores.

We observe that while the model trained on all features seems to perform best, the model trained on only a fraction of the features performs almost identically. The tables also show that if we were to take only one metric, then the number of equivalence classes is the best, unless Arjun is used as a preprocessor, in which

	SPUR	SPUR (+Arjun)	UniGen3	UniGen3 (+Arjun)
#mis	80.1	83.8	90.8	83.8
$\lvert E_F \rvert$	87.6	-	90.4	-
δ'	77.5	90.4	81.3	90.4
$\lvert Var(F) \rvert$	76.0	78.7	85.2	78.7
$\lvert F \rvert$	79.4	91.8	84.0	91.8
all	**89.2**	**95.2**	97.0	95.2
$\lvert Var(F) \rvert$, $\lvert F \rvert$, #l, tw, δ, #mis, $\lvert E_F \rvert$	**89.9**	**95.5**	97.7	95.5
$\lvert Var(F) \rvert$, $\lvert F \rvert$, tw, δ, #mis, $\lvert E_F \rvert$	88.6	94.5	**97.9**	94.5
$\lvert Var(F) \rvert$, tw, δ, #mis, $\lvert E_F \rvert$	**89.3**	94.5	97.7	94.5
$\lvert Var(F) \rvert$, tw, δ', #mis, $\lvert E_F \rvert$	88.1	**95.5**	**98.3**	**95.5**
$\lvert Var(F) \rvert$, δ', #mis, $\lvert E_F \rvert$	88.2	-	**97.7**	-
$\lvert Var(F) \rvert + \delta' +$ #mis	85.8	92.9	96.8	92.9
$\lvert Var(F) \rvert + \delta' + \lvert E_F \rvert$	87.8	-	94.2	-
$\lvert Var(F) \rvert + \delta'$	80.8	92.8	88.2	92.8
$\lvert Var(F) \rvert + \lvert F \rvert + \delta'$	79.5	93.7	88.2	**96.3**
$\lvert Var(F) \rvert + \lvert F \rvert +$ #l $+ \delta'$	80.4	93.5	88.7	**96.3**

Table 4.4: ROC AUCs with different features of a random forest containing 100 instances estimated using LOO

case δ' and the number of clauses seem to be very good candidates. If we focus on easily computable metrics, then the models that seem most promising are the ones trained on the number of variables, δ', and on the number of equivalence classes. If we preprocess the formulae with Arjun, then the number of variables combined with δ' seems sufficient. Furthermore, we find that using Arjun increases both F1 scores and ROC AUCs.

	SPUR	SPUR (+Arjun)	UniGen3	UniGen3 (+Arjun)
DT	82.4	**89.2**	**95.6**	94.8
RF 10	83.1	89.0	95.0	95.0
RF 100	**84.8**	88.5	95.4	**96.7**
RF 1000	83.5	88.3	95.2	96.5

Table 4.5: F1-scores with different models trained on $\lvert Var(F) \rvert$, δ' and $\lvert E_F \rvert$ estimated using LOO

In Table 4.5, we reported the F1-scores of decision trees (DT) and random forests (RF) using a different number of instances. The models were trained using $\lvert Var(F) \rvert$, δ', and $\lvert E_F \rvert$ (if Arjun is not used) and were evaluated using a leave-one-out strategy. We observe that a random forest containing 100 instances performs slightly better than the other models.

Answer to RQ2: We find that the number of equivalence classes alone forms an excellent predictor to classify sampling difficulty according to an affordability budget. Similarly, we observe that if Arjun is used to preprocess a formula, then prediction becomes easier.

4.4.3 RQ3: URS

	$\lvert Var(F)\rvert$	$\lvert F\rvert$	#l	tw	δ	#mis	$\lvert E_F\rvert$	Z3 time	Z3 mem
Z3 time	69.2	**72.5**	71.9	55.7	71.6	45.9	47.3	100.0	72.1
Z3 mem	76.9	**81.5**	79.9	64.7	80.9	49.5	56.5	72.1	100.0
MiniSAT time	68.0	74.2	74.0	64.6	**76.4**	45.8	55.3	64.3	73.4
MiniSAT mem	72.6	76.1	74.3	62.2	75.6	46.6	52.1	66.1	**80.8**
BSAT time	**77.7**	74.9	75.5	55.5	72.3	67.5	65.4	60.5	66.4
BSAT mem	**89.1**	86.0	83.5	65.2	82.9	62.4	71.1	66.8	76.0
D4 time	54.9	55.8	56.1	45.2	55.9	59.4	**70.3**	39.5	48.1
D4 mem	**64.5**	63.8	62.3	49.9	62.5	58.3	62.9	47.9	56.8
sharpSAT time	49.3	50.0	51.3	37.9	50.4	60.6	**64.0**	34.9	41.0
sharpSAT mem	35.8	35.8	36.5	22.3	34.9	**49.2**	42.8	21.8	25.8

Table 4.6: Kendall rank correlation coefficients of the used metrics with Z3 and MiniSAT (488 data points), as well as BSAT using Z3 (488 data points), D4 (437 data points) and sharpSAT (416 data points).

Table 4.6 shows the Kendall rank correlation coefficients for the MiniSAT and Z3 SAT solvers as well as our implementation of BSAT using Z3 and the state-of-the-art model counters D4 and sharpSAT.

All the 488 formulae have been used for the lines involving Z3, MiniSAT, and BSAT, as all managed to be sampled in less than 5 hours and with less than 64GB of virtual memory. The BSAT algorithm seems strongly correlated with the size of the MIS as well as the number of equivalence classes, but it is even more correlated with the number of variables, clauses, and literals. BSAT shows similar correlation coefficients to SPUR and UniGen3 with respect to the size of the MIS and the number of equivalence classes. We do find that D4 and sharpSAT have very similar correlation coefficients to both SPUR and UniGen3. This would indicate that in practice, the complexity of model counters and uniform random samplers is very close.

For both MiniSAT and Z3, we observe a strong correlation with the number of variables, the number of clauses, and the number of literals. We do not, however, observe a high correlation with the MIS or the number of equivalence classes. The correlation coefficients seem to be very different between SAT solving and URS. On

the other hand, BSAT seems to be a combination of SAT solving and URS in terms of correlations.

Answer to RQ3: SAT solving and URS correlate with different metrics and are thus different tasks. SAT model counting seems to be very close to URS. BSAT seems to be a combination of both SAT solving and URS in terms of correlation.

4.4.4 Perspectives

Our results demonstrate that the number of equivalence classes in a formula has strong correlations with the computational complexity of sampling. This opens the perspective of increasing sampling efficiency by transforming the input formulae into an equivalent formula with fewer equivalence classes, similarly to Arjun.

We also revealed that, though less than the equivalence classes, the MIS also shows strong correlations with sampling complexity. Therefore, efficient ways to compute the MIS and project a formula onto its MIS would also increase URS efficiency. This is demonstrated through the usage of Arjun, which further confirms our results. Moreover, Arjun allowed the samplers to solve more instances and increased the performance of our prediction models.

4.5 Threats to Validity

As for any empirical study, there are several threats to consider.

Construct Validity

To assess the validity of our findings, we used the Kendall rank correlation coefficient on the existing and our new $|E_F|$ metrics. The Kendall rank correlation coefficient is non-parametric (and therefore agnostic to the data distribution) and was used in the past to establish a relationship between structural metrics and runtime measures [AMM22]. Regarding the evaluation of our random forest predictors, we used both F1-score and Receiver Operating Characteristics (ROC) in order to cope with different classification thresholds. The main reason for this is that we have highly imbalanced data, and both metrics react differently to imbalance.

External Validity

We cannot guarantee that our findings generalise to any formula and all tools in each category (sampling, solving, counting). The reason behind this is the lack of general understanding of the complexity of SAT-based tasks [GV21], which we aim to address with new metrics. To mitigate this threat, we selected a range of SAT formulae from two different sources. They come from SAT Benchmarks used for the evaluation of uniform samplers [CMV14; CFM+15; DLB+18] and feature models

representing configurable systems of various types and sizes [PAP+19; APC21]. In both FM and non-FM categories, formulae encode different types of models: electronic circuits, algorithmic problems, etc., for the former, and Linux kernels, Unix command line tools, or configuration tools [HNA+18] for the latter.

4.6 Conclusion

To understand the complexity of SAT-based uniform sampling, solving, and counting, we have proposed an efficient algorithm to compute the equivalence classes (E_F) of a Boolean formula F. This metric possesses two desirable properties that other structural metrics fail to have both: *i)* a strong correlation to the computation time and memory consumption **and** *ii)* its computation scales even on complex formulae, thanks to its ability to exploit parallel computing infrastructures. We showed that $|E_F|$ can accurately (ROC AUC scores $> 87\%$) predict if a formula F is going to be easy or difficult to sample uniformly.

Furthermore, we showed that preprocessing techniques like Arjun can not only improve the scalability of samplers but also make the performance predictions of said samplers easier and more accurate, further motivating the development of efficient preprocessing techniques for URS and model counting.

We also highlighted that $|E_F|$ helped understand where URS complexity stands compared to two other SAT-based tasks: solving and model counting. We found that, at least in practice, URS is closer to model counting than to SAT solving. On the one hand, this prevents the naive use of standard solvers as uniform samplers. On the other hand, it further motivates research at the intersection of model counting with uniform sampling [SM21]. We expect our metric as well as Arjun to play a role in this bidirectional relationship, *e.g.,*, supporting the development of new knowledge compilation techniques.

5

Exploring the Computational Complexity of Uniform Random Sampling and SAT Counting with Phase Transitions

In this chapter, we further investigate formula difficulty by examining synthetic formulae and phase transitions. Analysing synthetic formulae allows us to isolate and study the impact of specific formula metrics while controlling for other variables.

Contents

5.1 Introduction

In this chapter, we explore phase transitions in URS and #SAT to deepen our understanding of formula hardness. Phase transitions are abrupt changes in a system's properties due to small variations in a parameter. Phase transitions in SAT, #SAT, and URS are defined as a critical point in a parameter space (e.g., clause-to-variable ratio) where solver behaviour changes abruptly, often following an easy-hard-easy pattern — problems are easy to solve before and after the transition, but hard near the critical point [GRM20; GYX11]. For 3-SAT problems, Mitchell et al. [MSL92] observed an easy-hard-easy pattern in the clause-to-variable ratio. As the ratio increases, the SAT solver runtime remains low until it spikes near 4.25, then decreases again for higher ratios. This phenomenon, known as the SAT phase transition, has driven research into algorithms focused on instances at this critical point [GLY17; DD06; PJM19; CLS14; MH15], highlighting its theoretical and practical significance [GW94]. Gupta et al. [GRM20] studied phase transitions for knowledge compilation and found that the phase transition for knowledge compilation occurs at a different clause-to-variable ratio than for SAT solving. However, these existing works did not study phase transitions for URS. Moreover, they did not study the effects of community structure on the phase transition, which is known to occur in real-world formulae [AGL12].

In this chapter, we contribute to a principled understanding of URS and #SAT complexity by studying whether phase transitions also occur in these problems. Our investigations require both experimental analysis (based on controlled experiments and artifacts) and empirical analysis (uncontrolled observations made on existing practices). While empirical observations on real-world formulae are needed to validate conclusions in practice, the low availability and high heterogeneity of these formulae entail an insufficient coverage of all possible structural variations to draw general conclusions. Therefore, we also conduct experiments on synthetic formulae created through systematic and controlled procedures to identify trends and limit cases. Doing so allows us to explore the role of specific characteristics in the complexity of URS and #SAT.

We begin by studying the complexity of model counters and uniform random samplers using k-CNF formulae, where each clause has exactly k literals. k-CNF formulae are commonly used in SAT studies [MSL92], enabling direct comparison with SAT solving and other knowledge compilation studies [GRM20]. We vary the clause-to-variable ratio, a key factor in phase transitions for SAT [MSL92], and set $k = 3$, as in previous SAT studies. We observe phase transitions in both URS and #SAT at a lower ratio than the SAT phase transition (2.00 vs. 4.25). This holds regardless of the formula's community structure [AGL12], though a weaker community structure leads to larger increases in computation time.

Next, similarly to [GRM20], we investigate the causes of the phase transitions.

While SAT phase transitions are linked to a sudden change in satisfiability probability, URS and #SAT, which explore all models rather than finding just one, require a different explanation. We find that the complexity of URS and #SAT can be explained by analysing the number of models relative to the number of variables, known as the solution density [BMC05; GRM20].

Finally, we empirically analyse real-world formulae to verify if the trends observed in synthetic data hold. Real-world formulae have a heterogeneous structure, as seen in Chapter 4, mixing clauses of varying sizes, which complicates general conclusions. Nevertheless, our observations suggest that phase transitions may not occur in real-world formulae — a finding that aligns with prior work by Mendonca et al. [MWC09], who report the absence of a SAT phase transition in formulae generated from feature models.

Altogether, this chapter makes the following contributions:

1. ***The first systematic study of phase transitions in both URS and #SAT.*** Our analysis of 2 samplers, 3 #SAT solvers, and 11,409 synthetic formulae shows that phase transitions occur in these problems, but at a different clause-to-variable ratio than in SAT. Thus, a formula that is easy for a SAT solver may not be easy for URS or #SAT.

2. ***A novel exploration of the reasons behind the complexity of URS and #SAT.*** Through our in-depth study of formula characteristics and the observation of phase transitions, we bring a fundamental contribution to the understanding of URS and #SAT complexity.

5.2 Objectives and Methodology

We detail below our research methods, the protocol to prepare the dataset (Boolean formulae), the samplers and model counters we used, and our computing infrastructure.

5.2.1 Research Questions and Methods

To achieve a principled understanding of URS and #SAT problem complexity, we explore the phenomenon of phase transitions [MSL92] similarly to SAT problems and knowledge compilation [GRM20]. Mitchell et al. [MSL92] showed that the solving difficulty of random k-CNF formulae changes drastically across the clause-to-variable spectrum. Indeed, they found that formulae are much harder to solve for a part of this spectrum. Around a clause-to-variable ratio of 4.25 (for $k = 3$), the formulae are exponentially harder to solve. This discovery has major importance as it shows that some of the synthetic formulae drawn from the same distribution are much harder to solve than others. This motivated the development of algorithms that perform better near the phase transition. Similarly, Gupta et al. [GRM20] investigated phase

transitions in knowledge compilation and identified a phase transition, though at a different clause-to-variable ratio compared to SAT solving. Even though phase transitions have been studied for #SAT and knowledge compilation [GRM20; HD04; GYX11; BL99], they remain a mainly observed phenomenon; as such, no formal definition is available. Because available real-world formulae are too scarce and heterogeneous to draw general conclusions (regarding structural properties such as the number of variables/clauses), our study combines experimental analysis (using synthetic artifacts under experimental control) with empirical analysis (based on real-world artifacts collected in the literature). These are necessary to draw theoretical (general) conclusions and validate them in practice.

Our study globally aims to answer three research questions:

- **RQ1:** Do phase transitions generally occur in synthetic URS and #SAT problems?
- **RQ2:** What are the reasons for the phase transitions in synthetic formulae?
- **RQ3:** Are phase transitions also observed on real-world formulae?

To draw general conclusions for RQ1 and RQ2, we rely on controlled experiments. To complement our findings with empirical observations, we answer RQ3 using real-world data.

5.2.2 Data Preparation

Synthetic Formulae

For our experimental analysis, we conduct controlled experiments with 11,409 formulae that we synthesise by varying multiple structural characteristics (number of variables, number of clauses, clause size, modularity, etc.). Thus, we can observe finely complex phenomena like phase transitions.

To generate our datasets, we use the classical k-CNF generation [MSL92] and the community attachment model from [GL15]. The classical model enables us to control various parameters: number of variables v, number of clauses n, and clause size k. It then generates a clause by selecting k unique variables and negating them with a $\frac{1}{2}$ probability, and adds the clause to the set of clauses. We use this model to generate a dataset containing 2,162 synthetic formulae, which will be used to answer RQ2.

The community attachment model from [GL15] gives additional control over the desired modularity Q of the formula. Controlling Q enables us to experiment on how it affects URS and #SAT time. The community attachment model requires specifying an initial number of communities c (groups of variables that are mostly dependent on each other but may have some dependencies with other communities). The generation then starts by splitting the variables into (roughly) equally-sized communities. It iteratively chooses to generate a clause with variables of a given community with probability $P = Q + \frac{1}{c}$; otherwise, all of the variables of the clause

to be generated will be part of distinct communities. We use this model to generate a dataset containing 9,247 formulae, which will be used to answer RQ1.

Real-World Formulae

We collected multiple datasets from Lagniez and Marquis [LM17], Soos [Soo24], Sundermann et al. [SBK+24], and Plazar et al. [PAP+19] to study whether our observations made on synthetic formulae also transfer to the real world.

The Lagniez and Marquis [LM17] 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 [Soo24] dataset contains 1896 formulae from various sources, including the Model Counting Competitions. The Sundermann et al. [SBK+24] 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. [PAP+19] is a dataset of 503 formulae consisting of a feature model benchmark (133 formulae) as well as other formulae collected from [DLB+18].

Overall, the dataset includes formulae featuring between five and 1,370,369 variables, and between ten and 5,798,978 clauses. The clause-to-variable ratio of the formulae ranges from 0.46 to 290.93 with an average value of 3.41 and a median value of 2.5.

5.2.3 URS and #SAT Tools

We selected diverse counters and samplers to explore phase transitions for different URS and #SAT algorithms.

We chose SPUR [AHT18] and UniGen3 [SGM20] as samplers as they both offer theoretical guarantees regarding their uniformity. Other sampling tools exist, *e.g.,* [DLB+18]. These alternatives offer no uniformity guarantee or rely on other tools such as sharpSAT [Thu06] or D4 [LM17]. We chose sharpSAT [Thu06] as a model counter because it is the fastest model counter to date [SHN+23]. We also chose D4 [LM17] as it is a model counter based on knowledge compilation and decision-DNNF, which is slightly different from sharpSAT. Moreover, D4 [LM17] will serve as a sanity check for our experimental setup because it has already been studied by Gupta et al. [GRM20]. We would like to highlight that a d-DNNF can be extracted from the execution trace of a model counter based on exhaustive DPLL. Therefore, we do not expect the results between D4 and sharpSAT to deviate significantly. Similarly, the results for SPUR and sharpSAT should be very close given their similarities. Finally, to have better diversity, we tested McTW [FHH20] because it is an algebraic-based model counter [SHN+23].

5.2.4 Infrastructure

We used a computing facility containing 354 nodes, each of which has 256 GB of RAM and 2 AMD Epyc ROME 7H12 CPUs running at 2.6 GHz. To measure the memory usage of the samplers, we developed a wrapper program that reads the appropriate file in the /proc folder, which contains information about the virtual memory usage of the program. We set a timeout of five hours and a limit of 64GB of virtual memory for counting and sampling experiments. Additionally, we requested 1000 models for each sampler.

5.3 Results

5.3.1 RQ1: Phase Transitions

To observe phase transition occurrences in URS and #SAT problems, we analyse the execution time of different sampling and counting tools on formulae systematically synthesised. Inspired by the work of Mitchell et al. [MSL92] on phase transitions in SAT problems, we aim to observe phase transitions on the clause-to-variable ratio (i.e. $|F|/|Var(F)|$) spectrum. All of our results are available on our companion GitHub [Zey25b].

Setup

We generate 3-CNF formulae defined over a set of 75 variables, with community structure. We set $k = 3$ because this clause size is typically used to analyse SAT problems, as it is the smallest non-trivial value for k. Indeed, $k = 1$ and $k = 2$ are not enough to create non-trivial dependencies between variables, whereas setting a value higher than 3 drastically increases sampling/solving time — as we also illustrate in RQ2. We set the number of variables to 75 because we experimentally determined this was a good trade-off for sampling/counting time (more variables reduce the impact of runtime noise in our results, but more than 75 was computationally prohibitive on our infrastructure).

Because we use 3-CNF formulae, we can compare our findings on URS and #SAT with the clause-to-variable ratio at the phase transition peak of SAT solving in [MSL92]. In the SAT experiments of Mitchell et al. [MSL92], solvers' execution time peaks at $|F|/|Var(F)| = 4.25$, and it corresponds to the ratio at which half of the 3-CNF formulae with this ratio are unsatisfiable. For lower ratio values (respectively higher), most of the formulae were satisfiable (respectively unsatisfiable). Since URS and #SAT on unsatisfiable formulae (or, more generally, on formulae with few models) can be efficiently reduced to one (or a few) call(s) to a SAT solver, we limit the clause-to-variable ratio to 10 (i.e., about twice the ratio at which SAT complexity is maximal). We also removed accidentally generated unsatisfiable formulae (i.e., with $|R_F| = 0$). Thus, we vary the number of clauses from 1 to 750

in steps of 1 (corresponding to a clause-to-variable ratio ranging from 1/75 to 10), and for each clause count, we generate five formulas to account for randomness in clause generation.

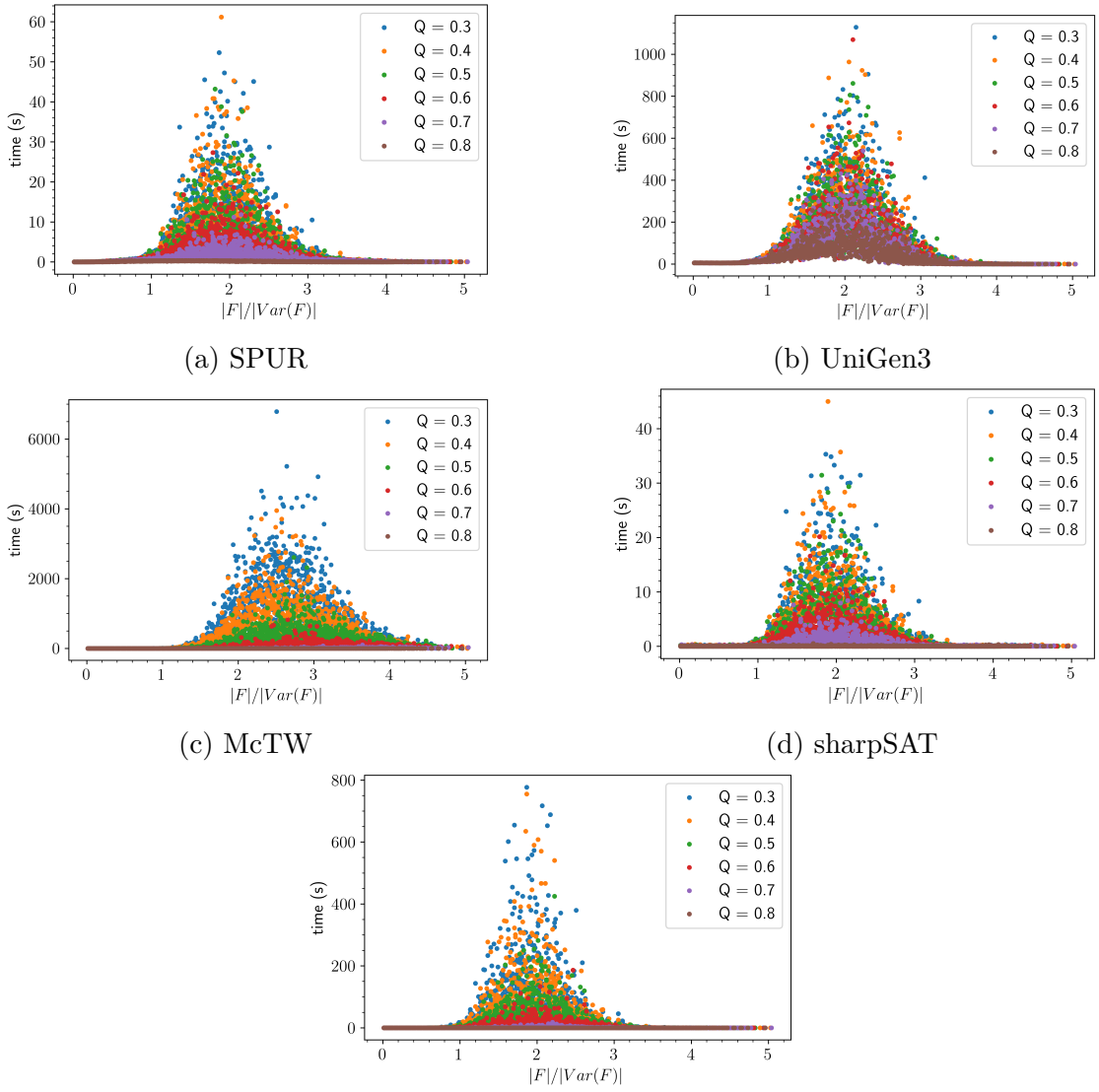
Finally, using the community attachment model from [GL15], we repeat the generation with different target modularity values Q (ranging from 0.3 to 0.8 with a 0.1 increment, as these values have been shown to be common in [AGL12] and in our experiments) to observe the impact of variable dependencies. This yields a total of 9,247 formulae. The community attachment model requires setting the desired number of communities. We choose to use 5 communities, as 15 variables per community should keep the probability of having an unsatisfiable community sufficiently low (considering that a community requires at least 2^k clauses on k variables to be unsatisfiable because a k-CNF formula expressed on k variables requires at least 2^k clauses to be unsatisfiable). For the sake of generality, our companion GitHub [Zey25b] page includes result plots for additional numbers of communities.

Results

Figure 5.1 shows the execution time for each URS and #SAT tool. The x-axis is the clause-to-variable ratio, while the y-axis is the execution time. Each data point represents a generated formula, and each colour corresponds to a different modularity value targeted by the generation algorithm. We only plot the values that have a clause-to-variable ratio in the range $[0; 5]$ because the formulae outside this range were unsatisfiable and therefore removed from the dataset.

We observe that phase transitions indeed occur for all tools (these are the 'peaks' we observe in each plot). Contrary to the SAT problem [MSL92], the hard instances are at a wildly different clause-to-variable ratio, i.e., around 2. URS and #SAT share the same clause-to-variable ratio for the hard instances, with the exception of the McTW model counter, whose performance is generally worse than other #SAT tools. The phase transitions that we observe align with the phase transitions observed in [GRM20; GYX11; HD04]. However, the phase transitions observed in [BP00; BL99] are shifted toward smaller clause-to-variable ratios, which may be due to better optimisations of the algorithms. A more surprising result is that the phase transition observed for UniGen3 is located at a similar clause-to-variable ratio than the phase transitions observed in our study and in [GRM20], even though UniGen3 has a different algorithm and a different purpose.

We also observe that lower target modularity coincides with a taller peak in execution time during the phase transition, indicating that tools exploiting modularity may better resist the complexity of URS and #SAT during the phase transition. A particular observation of UniGen3— which fundamentally exploits modularity less than other tools because the constraints added by the algorithm to the formula break the community structure — corroborates this finding: its



(a) SPUR

(b) UniGen3

(c) McTW

(d) sharpSAT

(e) D4

Figure 5.1: RQ1: Phase transitions occur in URS for 3-CNF formulae. A higher formula modularity decreases the height of the peak.

execution time peaks higher for high modularity values than the other tools. Similar results have been shown for SAT solving [GL15], where SAT solvers specialised in industrial formulae performed better on instances with a high modularity.

To complete these observations, we show in Figure 5.2 lower bounds to the actual modularity of the generated formulae across the clause-to-variable ratio spectrum. The colours indicate the target modularity parameter given to the community

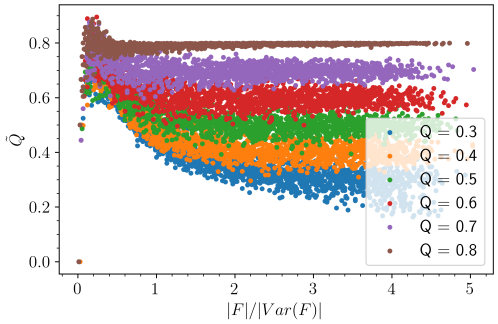


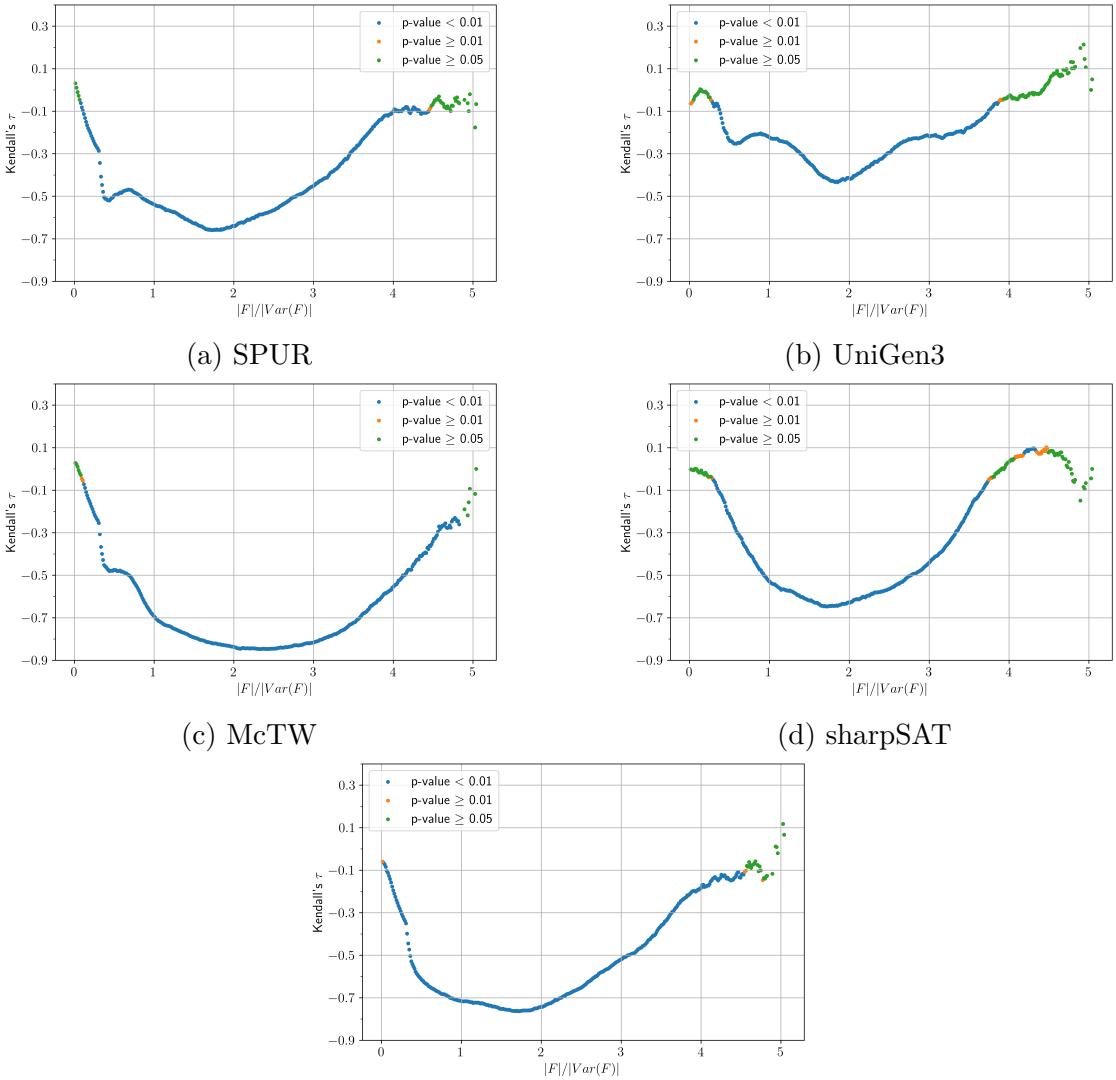
Figure 5.2: RQ1: Modularity of the 3-CNF formulae (y-axis) w.r.t. their clause-to-variable ratio (x-axis).

attachment model in [GL15], which may differ from the actual modularity given the heuristic-based nature of the model. The actual modularity values were computed by using the label propagation algorithm [AGL12] and are denoted by $\tilde{Q}$. This algorithm computes a lower bound for the actual modularity and, being based on label propagation, has stochastic components. Thus, to mitigate stochastic effects and obtain a bound close to the real modularity value, we repeat the modularity computation 100,000 times and take the highest value returned across all those runs.

Figure 5.2 shows that formulae with a low clause-to-variable ratio tend to have a high actual modularity (in particular, higher than the desired modularity specified in the generation algorithm), as is also shown in [GL15]. This is also one of the reasons why the phase transition does not occur at these lower clause-to-variable ratios.

Finally, to further confirm our results, we compute the Kendall rank correlation [Ken38] coefficients between the computed modularity $\tilde{Q}$ and the execution time of the different tools. We use the Kendall rank correlation [Ken38] because the relationship is unknown. Therefore, a rank correlation is better suited. Because of the observed phase transition, we find that computing Kendall's τ gives poor results. We thus decide to compute Kendall's τ over a sliding window. In other words, we compute Kendall's τ at a clause-to-variable ratio of x with the formulae that have a clause-to-variable ratio in the range $x \pm \varepsilon$. In our case, we set $\varepsilon = 0.3$.

The Kendall rank correlations are shown in Figure 5.3, where we plot the correlation coefficients across the different clause-to-variable ratios. The values shown in orange have a p-value higher than or equal to 0.01, and the values shown in green have a p-value higher than or equal to 0.05. We observe strong negative correlations across most of the clause-to-variable ratio spectrum with low p-values. This means that an increase in modularity is correlated with a decrease in computation time, as previously observed. Moreover, almost all of the samplers and



(a) SPUR

(b) UniGen3

(c) McTW

(d) sharpSAT

(e) D4

Figure 5.3: RQ1: Kendall's τ coefficients computed between $\tilde{Q}$ and the execution time in a sliding window across the clause-to-variable ratio spectrum.

model counters have strong correlations. The only sampler with lower correlation coefficients is UniGen3, which still shows a moderate correlation of -0.40, close to the observed phase transition. sharpSAT also exhibits a more surprising behaviour with positive correlations after a clause-to-variable ratio of 4. However, these correlations have high p-values. We thus conclude that community structure has a positive impact on the computation time of uniform random samplers and model counters.

RQ1 — Conclusions: Our results confirm the existence of phase transitions for URS and #SAT on 3-CNF formulae, which occur at different clause-to-variable ratios (starts at ≈ 1 and peaks at ≈ 2) than for classical SAT (peaks at 4.25). A higher formula modularity decreases the height of the peak. Formulae with a low clause-to-variable ratio (< 1) have a higher modularity, which might explain why URS and #SAT problems are easier to solve for those formulae.

5.3.2 RQ2: Reasons for Phase Transitions

We investigate the reasons behind the occurrences of phase transitions in URS and #SAT problems. In the case of SAT, the phase transition peaks at a 4.25 clause-to-variable ratio [MSL92], which corresponds to the ratio at which half of the 3-CNF formulae with this ratio are unsatisfiable. SAT solvers typically work by exploring a search tree where branches represent variable assignments [ES03]. Formulae with a low number of clauses per variable necessitate few assignments to be solved (it is easy to find a model), whereas formulae with a large number of clauses quickly lead to unsatisfiable branches (it is easy to conclude on the absence of models). At the ratio of 4.25, the SAT solver has to explore many deep branches to conclude the (un)satisfiability of the formula [MSL92].

By contrast, #SAT (and by extension URS, which relies on the same algorithmic principles) requires exploring all branches of the search tree to complete the counting, an exploration known as 'exhaustive DPLL'. Fortunately, the model counter can prune a part of a branch that has become trivial to solve. This can happen, e.g., if all of the constraints are satisfied by the current assignment a, in which case the current model count is $2^{|Var(F)|-|a|}$. Thus, the complexity of URS and #SAT is intrinsically linked with the minimal number of variables necessary to have at least as many models as the initial formula. In the above example, the minimal number of variables necessary to express the models in the branch is $(|Var(F)|-|a|)/|Var(F)|$.

Tool	$k=3$	$k=4$
D4	8.6	3239.97
sharpSAT	0.42	230.49
McTW	207.5	18000.0
SPUR	1.13	542.22
UniGen3	12.07	500.23

Table 5.1: RQ2: Maximum execution time (in seconds).

Gupta et al. [GRM20] generalise this idea and hypothesise that *what matters in knowledge compilation complexity is the solution density.* In other words, the

ratio of the minimal number of variables necessary to have at least as many models divided by the number of variables of the initial formula. We extend the hypothesis to URS and explore the ratio $r = log_2(|R_F|)/|Var(F)|$. The logarithm of $|R_F|$ expresses the minimal number of variables needed to encode $|R_F|$ models. Dividing this by the total number of variables allows us to have a normalised ratio.

We note that r is a semantic metric as it necessitates computing $|R_F|$, which may not always be feasible. However, our objective here is not to provide a metric to predict occurrences of phase transitions (the clause-to-variable ratio, which is always easy to compute, would be a more appropriate metric for that). Instead, we aim to study the phenomena behind phase transitions, and our findings (which corroborate Gupta et al.'s earlier investigations [GRM20]) indicate that r is a more effective metric for this purpose. Note that, since we exclude formulae with no model in our experiments, r takes a value between 0 and 1; moreover, we have $log_2(|R_F|) \leq |Var(F)|$ since $|R_F| \leq 2^{|Var(F)|}$.

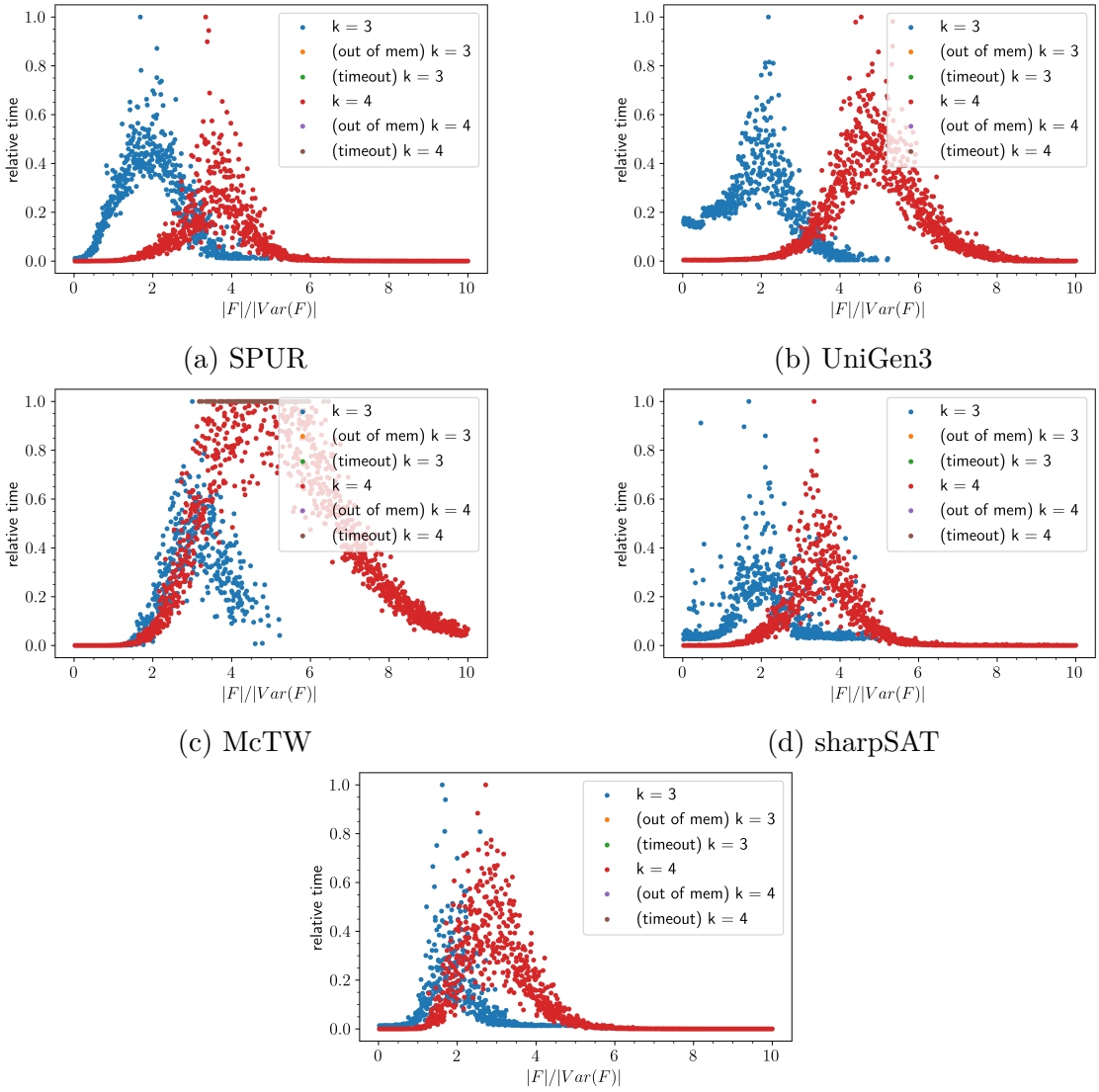
Setup

If our hypothesis is true, repeating our RQ1 experiments while setting $k = 4$ (instead of 3) should yield a shift in the observed phase transition. This is because for the same ratio $|F|/|Var(F)|$, increasing the number of literals in each clause yields a higher model count, as a larger clause is more likely to be satisfied by a random assignment. Thus, a higher clause-to-variable ratio should be necessary to observe the phase transition for $k = 4$ than for $k = 3$. To verify this, in addition to 3-CNF formulae, we also generate 4-CNF formulae. Because raising k from 3 to 4 significantly increases the URS and #SAT's tools execution times, we limit formulae to 50 variables.

With the above settings, we obtain 663 satisfiable formulae for $k = 3$ and 1499 formulae for $k = 4$ by using the k-CNF generation algorithm proposed by [MSL92]. We have more formulae for $k = 4$ because a higher clause-to-variable ratio is required to observe unsatisfiable formulae. To illustrate this increasing complexity, Table 5.1 reports the maximum execution time (in s.) of all tools across all formulae with $k = 3$ and $k = 4$; we observe that McTW has reached the timeout of five hours.

In Figure 5.4, we study the execution time of the URS and #SAT tools on all formulae with respect to their clause-to-variable ratio (x-axis). On the y-axis, we normalise execution time based on the maximum time taken by each tool for $k = 3$ and $k = 4$ separately, based on the values reported in Table 5.1. This normalisation helps us compare the trends for $k = 3, 4$, which would not be feasible otherwise due to large differences in execution time when increasing k.

In this figure, we observe that for $k = 4$ the phase transition indeed peaks at larger ratios for SPUR (3.5), sharpSAT (3.5), and UniGen3 (5), whereas for $k = 3$, the execution time peaks at a ratio of 2. There is also a large difference for McTW (3 vs 4.5). For D4, the two clause-to-variable ratios are closer to each other, though



(a) SPUR

(b) UniGen3

(c) McTW

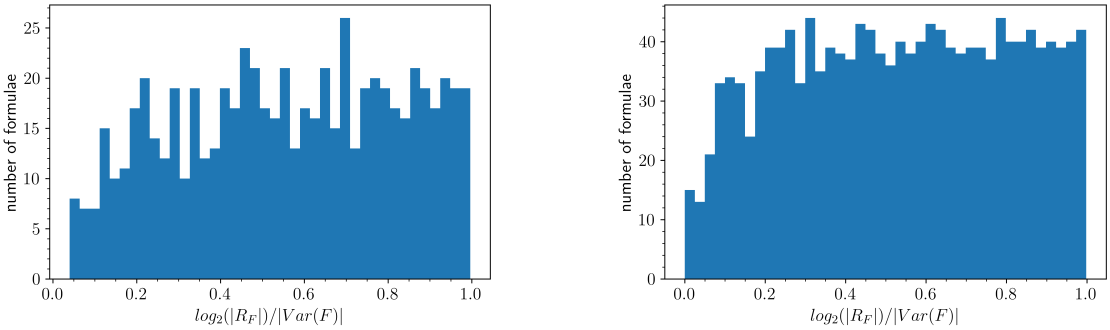
(d) sharpSAT

(e) D4

Figure 5.4: RQ2: Phase transitions w.r.t. the clause-to-variable ratio, on 3-CNF and 4-CNF formulae.

$k = 4$ observably yields a peak at a higher ratio. This observed shift is the first indication that our hypothesis indeed holds.

To confirm our hypothesis, we study the relationship between URS and #SAT execution time and the new ratio $r = \frac{log_2(|R_F|)}{|Var(F)|}$. We compute $|R_F|$ with D4 [LM17] for every formula. Because our formula synthesis algorithm is not driven by r or $|R_F|$, we first check that our population of formulae is sufficient to conduct analyses



(a) $k = 3$ (b) $k = 4$

Figure 5.5: RQ2: Distribution of $log_2(|R_F|)/|Var(F)|$ with respect to k.

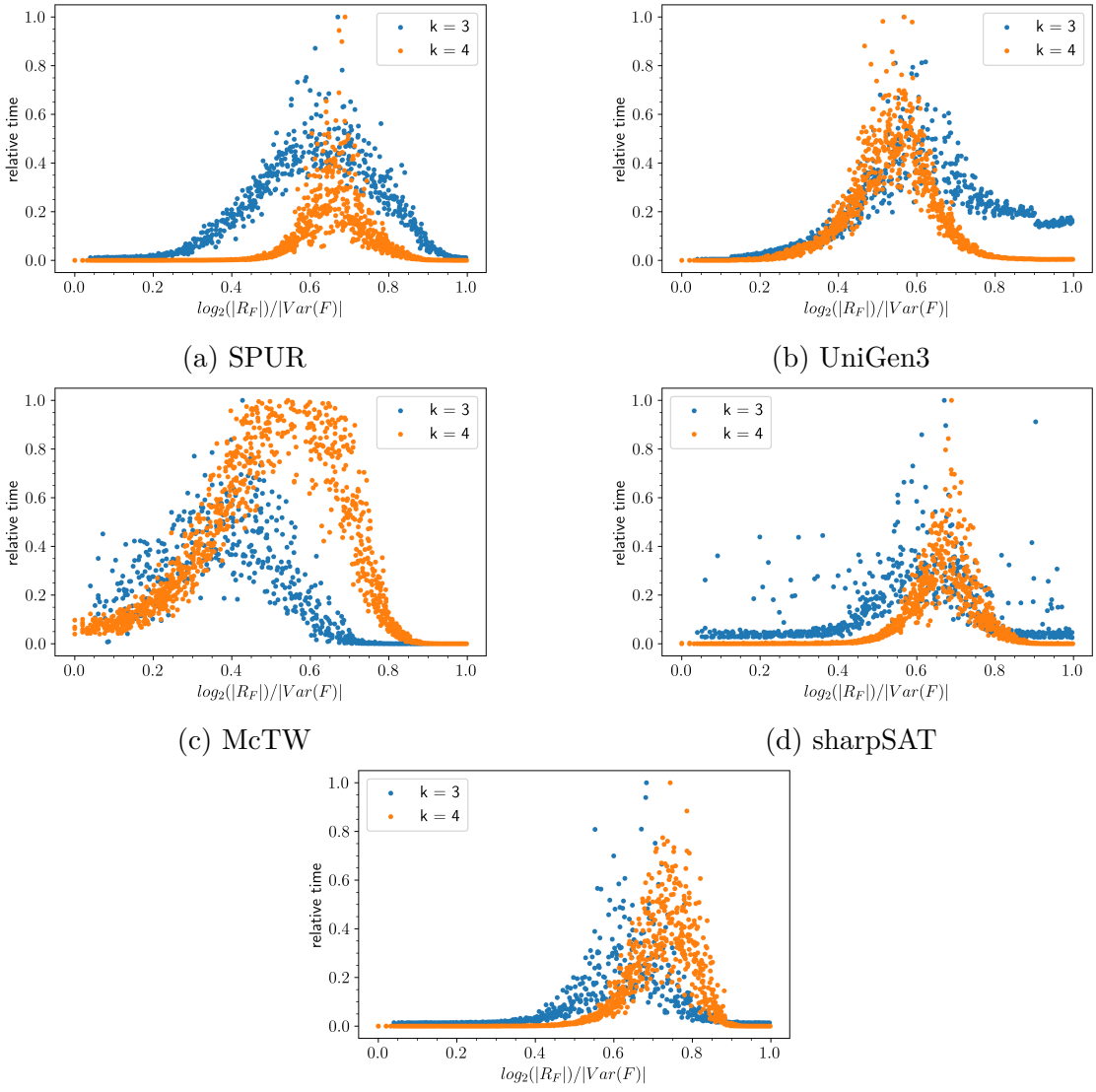
based on r (i.e., the formulae are sufficiently uniform with respect to r). Figures 5.5a and 5.5b show the distribution of r for the generated 3-CNF and 4-CNF formulae. We observe that the distribution is balanced, except for a small deficit when approaching $r = 0$. This can be explained by the fact that at such r values, a synthesised formula has a higher probability of being unsatisfiable. Fortunately, this bias in the population does not affect the phase transition, which should occur far from $r = 0$ (as later confirmed in our results).

Results

Figure 5.6 plots the tools' normalised execution time (y-axis). We observe that the hard instances for SPUR and sharpSAT occur at a ratio $r = log_2(|R_F|)/|Var(F)|$ close to 0.65 for both 3-CNF and 4-CNF formulae, whereas the value r for UniGen3 is between 0.55 and 0.6. As for D4 and McTW, we also observe a phase transition, though the peaks deviate slightly between $k = 3$ and $k = 4$ (from 0.68 to 0.74 for D4, similarly to [GRM20], from 0.40 to 0.55 for McTW).

To evaluate whether r provides a better characterisation of the phase transition than the clause-to-variable ratio, we compute the **mean absolute error (MAE)** in each case. As a reference, we define a function $f(x)$, computed on the dataset corresponding to $k = 4$, where each value x is associated with the mean of all data points within the interval $x \pm \varepsilon$. Specifically, we use $\varepsilon = 0.3$ when analysing the clause-to-variable ratio, and $\varepsilon = 0.028$ when analysing r. These values of ε were chosen to reduce noise-induced fluctuations while preserving the overall trend of the data. In both cases, the size of the sliding window was adjusted to include, on average, the same number of data points — approximately 86 ± 1.

The function $f(x)$ is then normalised to lie within the interval $[0, 1]$. Finally, we compute the MAE between the normalised reference function $f(x)$ and the corresponding dataset to compare the performance of r and the clause-to-variable ratio in describing the phase transition.



(a) SPUR

(b) UniGen3

(c) McTW

(d) sharpSAT

(e) D4

Figure 5.6: RQ2: Phase transitions w.r.t. $r = log_2(|R_F|)/|Var(F)|$ on 3-CNF and 4-CNF formulae.

Table 5.2 reports the computed MAEs for both the clause-to-variable ratio and the parameter r, for $k = 3$ and $k = 4$. The left half of the table corresponds to the results shown in Figure 5.4, while the right half relates to Figure 5.6.

The MAE values for $k = 4$ serve as a sanity check, as the reference function $f(x)$ is constructed from the $k = 4$ dataset. We observe that for $k = 3$, the MAE consistently decreases when switching from the clause-to-variable ratio to

Tool	clause-to-variable ratio $k=3$	clause-to-variable ratio $k=4$	r $k=3$	r $k=4$
D4	0.368	0.109	0.175	0.109
sharpSAT	0.353	0.113	0.158	0.110
McTW	0.185	0.054	0.299	0.057
SPUR	0.377	0.116	0.138	0.117
UniGen3	0.314	0.119	0.184	0.119

Table 5.2: RQ2: Mean absolute errors of the datasets with the $k = 4$ dataset as reference.

r, suggesting that the apparent shift between the curves for $k = 3$ and $k = 4$ is reduced when using r. This supports the conclusion that r more accurately aligns the phase transitions across different values of k.

The only exception is observed in the case of McTW, where the MAE increases when switching from the clause-to-variable ratio to r. We attribute this deviation to the large number of timeouts encountered by McTW, which may affect the accuracy of the underlying data.

Overall, the results indicate that r provides a better description of the phase transition than the clause-to-variable ratio.

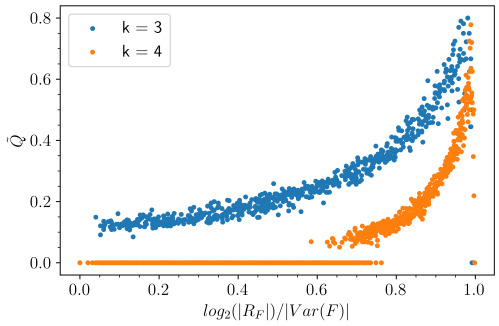


Figure 5.7: RQ2: Modularity w.r.t. $r = log_2(|R_F|)/|Var(F)|$.

Discussion Our observations seem to indicate that the hard instances of URS and #SAT need model spaces R_F that are large but with sufficient differences between each model (in terms of assignments). In other words, instances with a high r (equivalently, a low clause-to-variable ratio) are easy to solve because they are underconstrained and quickly generate unconstrained (free) variables during the DPLL exploration, making parts of some search branches trivial to count. By contrast, instances with a low r (or high clause-to-variable ratio) are overconstrained, and their model count is low, which is, therefore, easy to enumerate and count. The

instances in between do not allow for an effective processing because the models are too different from each other, while the large $|R_F|$ prohibits model enumeration.

A complementary explanation relates r to modularity. Figure 5.7 shows the actual modularity of the 3-CNF and 4-CNF formulae across the r spectrum. As before, we compute modularity with the label propagation algorithm in [AGL12] repeated 100,000 times and took the maximum modularity value (maximum lower bound). We observe that a high ratio r coincides with a high modularity. This indicates that we have a strong community structure in the formulae and that the solvers would need to satisfy fewer clauses before having entirely disjoint communities. These disjoint communities can then be processed independently, which is D4's strategy [LM17]. Lower values of r have a low modularity as these formulae are overconstrained.

RQ2 — Conclusions: Phase transitions in URS and #SAT are better described by the ratio of variables required to encode the space of models. Formulae with a high ratio are easy to solve because they are underconstrained, enabling URS and #SAT to take shortcuts and consider unconstrained variables separately. Conversely, those with a low ratio r have few models that are fast to enumerate. The phase transition is due to intermediary situations wherein the number and diversity of models impede both model counting and effective solving.

5.3.3 RQ3: Real-World Formulae

Setup

Our last RQ analyses real-world formulae. Unlike synthetic k-CNF formulae, real-world formulae are likely to include constants and variables that do not appear in clauses (this typically happens, e.g., to formulae derived from feature models because some features may be mandatory and other features may not have any constraints) [LGC+15]. To avoid noise in our computed ratios, we first pre-process the formulae to eliminate these constants and unconstrained variables (without altering the formulae's semantics). This modification does not affect our analysis of the tools' execution time since the underlying algorithms also remove them during their pre-processing. Furthermore, we also remove redundant clauses, i.e., clauses that are subsumed by another. Assume we have $c_i, c_j \in F$ with $i \neq j$ and $c_i \implies c_j$. Then we only need to consider c_i and can safely ignore c_j because c_i ensures that c_j is satisfied. We thus compute the size of the formula $|F|$ as the number of clauses minus the number of redundant clauses. $|Var(F)|$ is computed as the total number of variables minus the number of variables that do not appear

in any (non-redundant) clause. Both the URS and #SAT tools were executed on the original formulae.

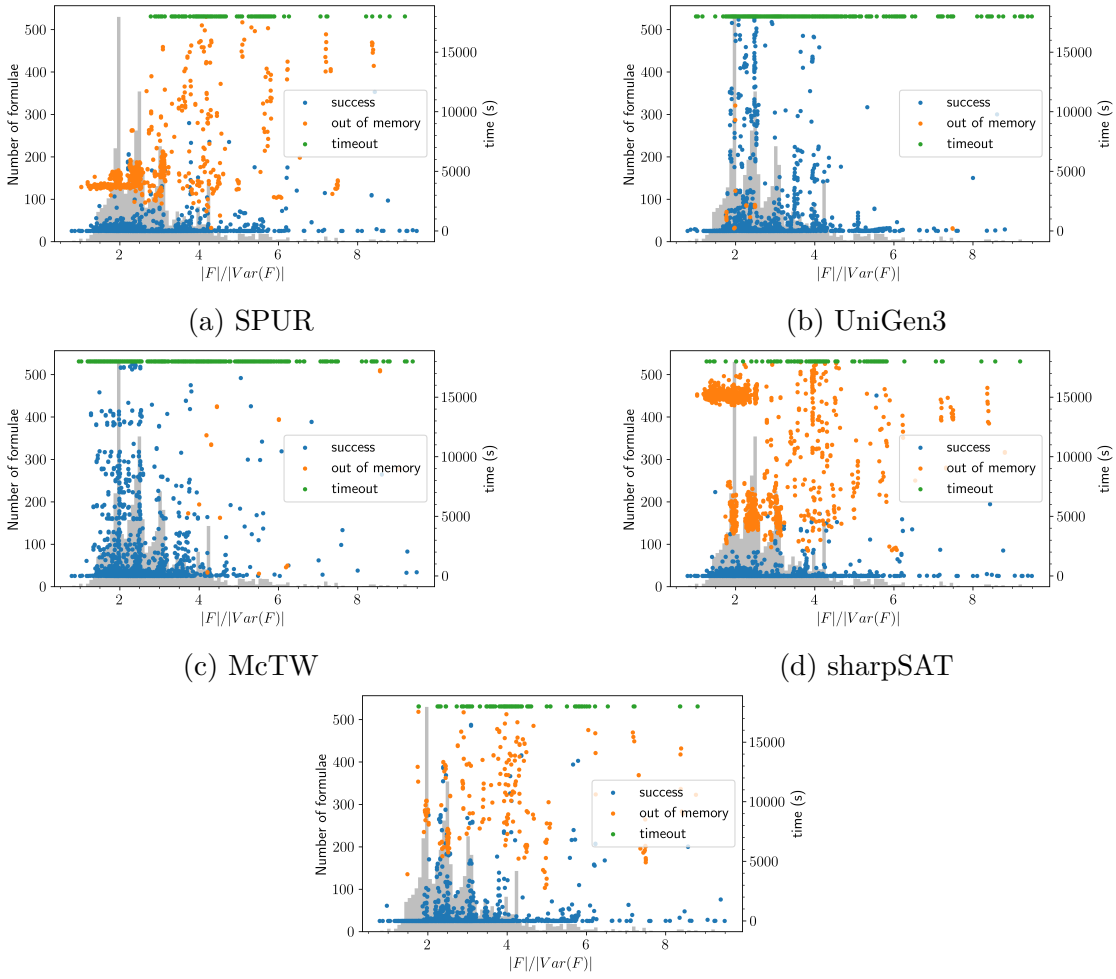
Our statistics on the capacity of the URS and #SAT tools to analyse real-world formulae indicate that URS and #SAT tools have approximately the same success rate with respect to timeouts and out-of-memory exceptions. UniGen3 performed significantly worse, with only 2510 formulae that were successfully processed. The other tools processed with success between 3034 and 3996 formulae, with McTW processing successfully 3034 formulae and D4 processing successfully 3996 formulae.

Results

Figure 5.8 shows the execution times of the different model counters and samplers on our real-world formula dataset. The x-axis denotes the clause-to-variable ratio. The reason we use this ratio and not r is that model counting (which is required to compute r) is intractable for a significant number of formulae in our dataset; discarding these formulae would leave us with a smaller number of formulae to conduct our analysis. Additional Figures using r are available on our companion GitHub [Zey25b].

The double y-axis reports two measures: the number of real-world formulae (left-hand side scale) and the execution time (right-hand side scale). Thus, the grey histogram shows the frequency of each clause-to-variable ratio among the set of formulae, while the coloured dots represent the execution time of each formula on the considered tool. The colours distinguish the formulae that were processed with success (blue), the formulae for which the model counter or sampler ran out of memory (orange), and the formulae for which the model counter or sampler reached the 5-hour timeout (green).

In Figure 5.8, we see that while there are some clusters, overall, patterns for phase transitions are difficult to observe. To complement this view, we show in Figure 5.9 the cumulative distribution of the total number of formulae, the number of formulae processed with success, and the number of formulae that could not be processed with respect to their clause-to-variable ratio. While there do seem to be spikes in the line representing failure (the green line), these seem to align with the total number of formulae as shown in the histogram in Figure 5.8. Generally, the absence of clearly observable phase transitions, coupled with the significant number of formulae intractable for URS, indicates that there are additional factors in feature-model formulae that explain their hardness. This means that specific applications of URS to feature models necessitate dedicated complexity studies, as results valid for synthetic formulae are not enough to explain this complexity. Furthermore, the lack of a phase transition for URS and #SAT in real-world formulae is consistent with the findings of Mendonca et al. [MWC09], who did not observe a phase transition for SAT in feature models.



(a) SPUR

(b) UniGen3

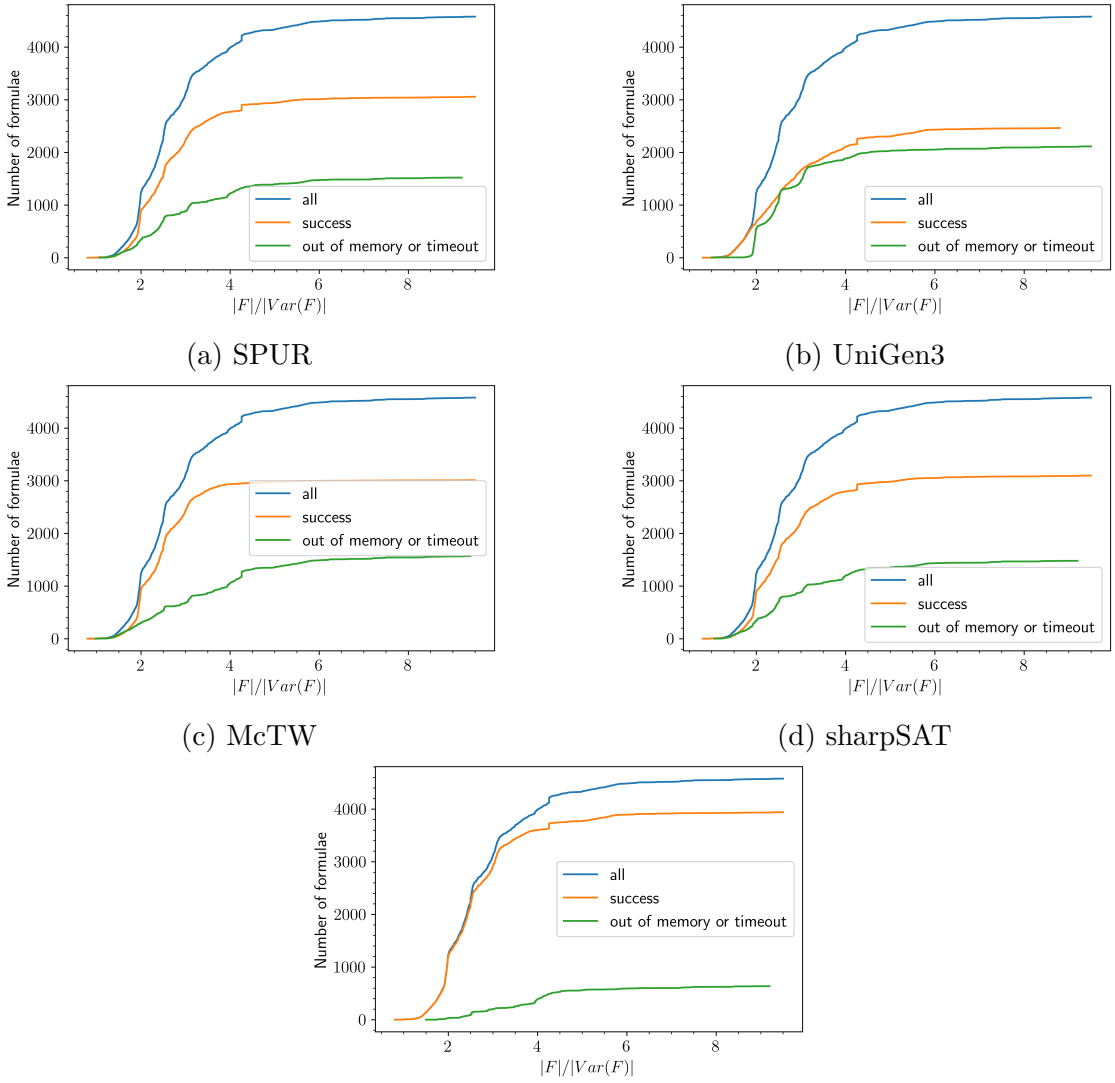
(c) McTW

(d) sharpSAT

(e) D4

Figure 5.8: Results on real-world formulae.

RQ3 — Conclusions: In the case of feature models, the observation of phase transitions is blurred by additional complexity factors that do not occur in synthetic formulae. This calls for novel complexity studies specific to feature models and dedicated methods to decompose this complexity in a form that makes it tractable for URS.



(a) SPUR

(b) UniGen3

(c) McTW

(d) sharpSAT

(e) D4

Figure 5.9: CDF of real-world formulae wrt. $|F|/|Var(F)|$.

5.4 Threats to Validity

Internal Validity

This threat concerns the implementation and the choice of specific parameters during our experiments. We chose to execute our experiments on random formulae with 50 and 75 variables, which is fewer than most real-world formulae. This was necessary to execute this large synthetic benchmark on multiple solvers and avoid long executions and timeouts. Yet, our experiments showed that synthetic formulae

are much more difficult to count or sample from than some real-world formulae with more variables. Regarding the community attachment model, we set the number of communities to 5. This may introduce a bias towards formulae with specific modularity. Thus, we reran the experiments with different numbers of communities and provided the results on our companion GitHub [Zey25b].

External Validity

There is no guarantee that our results generalise precisely to any formula and any model counter or sampler in each category. The reason behind this is the lack of general understanding of the complexity of SAT-based tasks [GV21], which we aim to address. To mitigate this threat, we selected a range of SAT formulae from multiple sources. They come from SAT Benchmarks used for the evaluation of uniform samplers [CMV14; CFM+15; DLB+18] and various other sources such as feature models representing configurable systems of various types and sizes [PAP+19; APC21; SBK+24]. The datasets that we use include formulae that encode diverse types of models: electronic circuits [LM17], algorithmic problems, Linux kernels [PAP+19; SBK+24], Unix command line tools, or configuration tools [HNA+18]. Thus, we are confident that our general conclusions are valid for a large class of real-world formulae.

5.5 Conclusion

In this chapter, we analysed the experimental complexity of counting and uniform random sampling for Boolean formulae, under the prism of phase transitions and community structure. Our investigation initially focused on synthetic formulae, allowing us to finely explore phase transitions and the role of community structure in a controlled way. Hence, we demonstrated that phase transitions indeed occur in URS and #SAT, while the community structure impacts the amplitude of the peak. We also showed that phase transitions are harder to observe in real-world formulae, although many of these formulae remain intractable. We therefore believe that additional complexity factors are at play and need feature-model-specific studies to be explained. Overall, our work contributes to a principled understanding of URS and #SAT complexity, and we hope it can inspire future research in designing effective methods to approach these problems.

6

Testing Uniform Random Samplers: Methods, Datasets and Protocols

Demonstrating that a uniform random sampler truly produces uniform samples can be challenging. Furthermore, such proofs typically apply to the algorithms rather than their specific implementations. Consequently, it is important to empirically test uniform random samplers. In this chapter, we introduce five statistical tests specifically designed for evaluating the uniformity of random samplers.

Contents

6.1 Introduction

The key role of URS in its applications is that uniformity enables unbiased sampling from a large model space. Some samplers provide theoretical guarantees of uniformity, thereby proving that their algorithm generates samples with uniform probability [AHT18; SGM20]. However, these proofs of uniformity are hard to provide for multiple reasons. First, some samplers favour practical heuristics that trade off potential theoretical guarantees for improved efficiency [EGS12]. Second, the theoretical guarantees are provided for the algorithm and not for the implementation of an algorithm, which may break the theoretical guarantees due to the way the sampler is implemented, or merely if the implementation contains undetected bugs.

In this chapter, we seek to establish a methodological ground for testing sampler uniformity in practice by means of a set of statistical tests. Our contributions not only enable researchers to compare practical uniformity confidence across multiple sampler candidates, but they can also be useful to detect implementation mistakes leading to non-uniform tools, although based on a theoretically uniform method. The statistical tests composing our method constitute an actionable solution for URS researchers to assess the uniformity of novel approaches while removing the necessity of developing arbitrarily complex (and, as argued before, only partially reliable) theoretical proofs. All in all, our contribution aims to foster and accelerate research in URS by automating empirical testing and enabling comparison with a wide range of existing samplers.

To achieve this, we inspire ourselves from the pseudo-random number generator (PRNG) community [LS07; Dot; BEB18], which has solid experience in practical testing approaches. Unfortunately, we cannot reuse these PRNG tests because the model space of a PRNG is unconstrained, unlike Boolean formulae in URS. Nevertheless, we follow the good practices of the PRNG community to establish our own test protocol. In particular, this community has long used multiple statistical tests to test their tools, as each test has different strengths and weaknesses [RSN$^+$01]. Relying on a single test A is bad practice, as a PRNG could be engineered to specifically pass test A without necessarily producing high-quality pseudo-random numbers.

Applying this lesson learned from the PRNG community, we start by providing a statistical test suite that is designed to test the uniformity of uniform random samplers. These tests have been selected and adapted from the available literature to form a consistent testing approach for URS. After describing the different tests that make up our suite, we apply them in an empirical study involving the available URS tools. Throughout this study, we can not only compare the uniformity test results for the samplers but also reveal the consistency of all tests (or lack thereof) on a given sampler. Through these investigations, we confirm the existence of

an experimental bias in the choice of a specific test and, thereby, the necessity of conducting different tests.

Through our applications of our tests to state-of-the-art URS tools, we find that different tests not only have different reliability but also come with varying computational costs. Based on these insights, we suggest a testing process that executes different tests in a sequence such that tests with a lesser computational cost — and a potentially higher Type II error rate[1] — are executed first. We also indicate which of these tests reveal less information regarding sampler uniformity and can be omitted in the case of constrained computational resources.

Next, we explore another threat to the validity of uniformity testing: the input Boolean formula used for testing. Indeed, while one could consider uniformity as a universal property of sampling methods, the empirical nature of both sampler implementation and uniformity tests creates an inherent risk for a test to generate Type I and Type II errors. We, therefore, consider the use of multiple formulae for uniformity testing and present a methodology to combine results for individual tests into a statistically meaningful answer. Beyond this, we study the question of dataset bias. In particular, we investigate how test results obtained from synthetic formulae typically used in the URS research community correlate with results on real-world formulae extracted from feature models. Establishing these correlations (or their absence) would determine the importance and limitations of synthetic datasets in uniformity testing.

To summarise, this chapter makes the following contributions:

1. **A uniformity testing procedure.** Our method includes five tests, of which three have never been applied as statistical tests for sampling uniformity. Our procedure also considers the relative computational costs of the tests in order to quickly eliminate non-uniform samplers and limit the execution of more expensive tests.

2. **Uniformity results on state-of-the-art samplers.** We report on a large experimental study studying the uniformity of state-of-the-art samplers according to our different tests. We reveal that among our tested samplers, only UniGen3 is uniform, unlike what previous studies concluded. These results show the importance of conducting multiple independent tests to ensure reliability, while also avoiding the pitfall of previous approaches in using a supposedly uniform sampler as a reference [SGC+22].

3. **Insights into dataset choice.** We reveal that URS tools that fail to sample uniformly from synthetic formulae are also unlikely to produce uniform

[1]Our null hypothesis is that a given sampler is uniform. Therefore, a Type I error refers to the rejection of this hypothesis while the sampler is uniform. Type II error refers to the conclusion that a sampler is uniform when it is not.

samples from real-world formulae. The contraposition, however, does not hold, revealing the need to benchmark samplers against a diverse set of real-world formulae.

6.2 Statistical Test Methodology

Our objective is to provide a statistically grounded method to test the uniformity of URS tools. The key assumption of our method is that uniformity testing typically suffers from two pitfalls: the reliance on a single, specific test and the biases introduced by the choice of the input formula. Given these pitfalls, our approach advocates the use of several statistical tests and the combination of results obtained from different formulae into a single statistical answer.

6.2.1 Combining Results from Multiple Formulae

In contrast to testing pseudo-random number generators (PRNGs), testing samplers also requires a formula to be given as input to the sampler under test. To alleviate possible biases due to the input formula, we apply each statistical hypothesis test (aka statistical test) described below to multiple formulae. Unless otherwise stated, by the term *test* we refer to the particular application of a given statistical test to a given sampler using a given Boolean formula. Thus, for a given statistical test and sampler, we conduct one test per available formula.

Each test produces a single p-value, which can be used to make inferences. The family-wise error rate (FWER) is the probability of making at least one Type I error when performing multiple tests. The probability α of making a Type I error is the probability of rejecting the null hypothesis H_0 when H_0 is true. Or in our case, the probability of wrongly rejecting the uniformity of a sampler even though the sampler is uniform.

The probability of making a Type I error if we perform N tests is computed as follows:

$$\alpha' = 1 - (1 - \alpha)^N$$

with α the significance level of a single test. Thus, α' increases as the number of tests N (in our case, formulae) increases. This is undesirable as it may lead us to wrong conclusions about the sampler under test. If we set $\alpha = 0.01$ and $N = 20$ we find $\alpha' = 0.18$. This means that we have a probability of 0.18 (if H_0 is true) that at least one test will reject the null hypothesis, which may lead us to a wrong conclusion. Considering our research questions, this issue needs to be addressed. There exists a multitude of controlling procedures to alleviate this issue.

Bonferroni Correction

The Bonferroni correction is a classical way of controlling the FWER. If N tests are performed and we want to ensure a probability of performing a Type I error

of at most α, then we set $\alpha_i = \frac{\alpha}{N}$ with α_i the significance level of the individual tests. The Bonferroni correction thus ensures that $\alpha' = 1 - (1 - \alpha_i)^N \leq \alpha$ without imposing any assumptions about the dependence between the p-values.

Harmonic Mean p-value (HMP)

The harmonic mean p-value (HMP) [Goo58; Wil19] assesses the significance of groups of hypothesis tests while also controlling the strong-sense family-wise error rate. The Harmonic mean p-value improves on the power of the Bonferroni correction. Additionally, the HMP does not require the p-values to be independent.

The harmonic mean p-value is defined as follows:

$$\mathring{p}_R = \frac{\sum_{i \in R} w_i}{\sum_{i \in R} \frac{w_i}{p_i}} \tag{6.1}$$

With R any subset of the m tests, w_i the weights of each test, $\sum_{i=1}^{m} w_i = 1$ and, p_i the p-value of each test. To perform a family of tests at a significance level of approximately α, we reject the null hypothesis that none of the p-values in the subset R are significant when $\mathring{p}_R \leq \alpha w_R$ (with $w_R = \sum_{i \in R} w_i$). The approximation is reasonable for a small significance level α and improves with smaller values. In other words, if we find $\mathring{p}_R \leq \alpha w_R$ then we can safely assume that at least one of the tests was significant and thus reject the null hypothesis with a significance level of approximately α.

In this chapter, we use the HMP. We perform the desired statistical test on a set of formulae on a given sampler. This gives us a vector of p-values on which we compute the HMP $\mathring{p}$. We may then compare $\mathring{p}$ to the appropriate significance level α. If we find $\mathring{p} \leq \alpha$ (because we compute $\mathring{p}$ on all the p-values we have $w_R = 1$), then we know that at least one of the tests is significant and we can reject H_0. Otherwise, we fail to reject H_0.

6.2.2 Statistical Tests for Uniform Random Sampling

We propose five statistical tests to empirically assess sampler uniformity. The intuition behind the use of multiple tests is that each test assesses whether a particular statistical value from the model space is preserved in the sample. Therefore, the result for a single test is bound to the statistical value it evaluates, which makes it necessary to execute multiple tests to obtain higher confidence regarding sampling uniformity. To build the tests, we consider the following hypotheses:

- H_0 (null hypothesis): the sampler samples uniformly,
- H_a (alternative hypothesis): the sampler does not sample uniformly.

Pearson's χ^2 Statistical Test (GOF)

The process of examining how well a sample agrees with a probability distribution is known as a *goodness of fit* test. We describe below how one can apply Pearson's

χ^2 test [Pea00] to uniform random samplers.

Pearson's χ^2 test requires two vectors E and O. The vector E contains the expected data under H_0, and O contains the observed data of a sample S with $S \in {R_F}^N$, the tuple containing the sampled models.

We define $O = (O_1, O_2, ..., O_{|R_F|})$ for a sample S as $O_i = |\{m \in S | id(m) = i\}|$. With id, a bijective function $id : R_F \to \{1, 2, ..., |R_F|\} \subseteq \mathbb{N}$. We follow by defining $E = (E_1, E_2, ..., E_{|R_F|})$ as $E_i = \frac{N}{|R_F|}$. This follows from the definition of H_0. As we want uniform probability, we expect the same number of occurrences for every model.

Pearson's χ^2 test statistic is defined as follows:

$$\chi^2 = \sum_{i=1}^{|R_F|} \frac{(O_i - E_i)^2}{E_i} \tag{6.2}$$

The p-value for Pearson's one-sided χ^2 test (with $|R_F| - 1$ degrees of freedom) can be derived from the statistic. The p-value is then compared to a desired significance level α. If the p-value is smaller than α, H_0 is rejected and H_a is accepted. Otherwise, no clear conclusion can be reached, and we fail to reject H_0. In our case, we will simplify the conclusion by accepting H_0.

The main disadvantage of the test is that it works with the entire probability distribution. Thus, it requires an array of elements representing the probability distribution. This means one value for each possible output. In the case of a PRNG, this means having an array of size 2^{32} or 2^{64} in the case of 32-bit or 64-bit PRNGs, respectively. In the case of Boolean formulae, the problem is similar; the hyperspace is often too large, which prohibits us from allocating an array of the required size, as a lot of formulae have more than 2^{64} models. Moreover, the GOF test is known to be unreliable if any expected frequency is below five [Yat34]. In other words, in the case of uniform random sampling, the test requires us to sample at least five times the total number of models to get reliable results. Thus, GOF only scales to small hyperspaces.

Variable Frequency (VF) Statistical Test

Plazar et al. [PAP+19] developed a test in which they compare the expected variable frequencies with the observed variable frequencies of the sample. We extend their work by replacing the fixed threshold with a GOF test. This allows us to have more reliable results.

A uniform sample is expected to be representative of the set of models. Thus, the observed variable frequencies should not deviate significantly from the expected frequencies. The VF test only tests that the observed variable frequencies are representative of the real variable frequencies. While this is not enough to certify true uniformity, it may be enough depending on the use case. As an example,

suppose that a variable a is of interest because the software component a has a bug. If we know the variable frequency of a, then we have an idea of how widespread the problem is relative to the entire software product line.

Next, we show how to perform the VF test. The resulting test is similar to the test performed by Plazar et al.

To test our sampler, we sample N models. Let $S \in {R_F}^N$ be the tuple containing the sampled models. We then perform a frequency test for each variable. We ignore variables that are always true or always false because any sampler would generate the correct distribution for these variables as long as the sampled assignments are models of the formula under test.

To test the individual variable frequencies, we perform Pearson's χ^2 test on the vectors O and E defined as follows. We define $O_{v=1} = |\{m \in S | v \in m\}|$ (resp. $O_{v=0} = |\{m \in S | v \notin m\}|$) as the number of sampled models with the target variable v set to true (resp. false). We define $E_{v=1} = \frac{N}{|R_F|}|\{m \in R_F | v \in m\}|$ (resp. $E_{v=0} = \frac{N}{|R_F|}|\{m \in R_F | v \notin m\}|$) as the expected number of models with the target variable set to true (resp. false) under H_0.

We compute Pearson's χ^2 test statistic for each variable as follows:

$$\chi^2_v = \sum_{i \in \{1,0\}} \frac{(O_{v=i} - E_{v=i})^2}{E_{v=i}} \tag{6.3}$$

We follow by deriving individual p-values p^F_v from each one-sided test (with one degree of freedom). Unfortunately, a direct comparison with a significance level α is impossible, as performing multiple tests raises the family-wise error rate (FWER), i.e., performing multiple tests raises the probability of rejecting H_0 even though H_0 is true (Type I error). To mitigate this, we use the harmonic mean p-value defined in Subsection 6.2.1. We thus compute all the p-values p^F_v associated with every variable v of F (that satisfies $E_{v=0} \neq 0 \wedge E_{v=1} \neq 0$) and compute $\mathring{p}_F$ as shown in equation 6.4 on the entire set of p-values (we give equal weights to all the p-values).

$$\mathring{p}_F = \frac{\sum_{v \in \Gamma_F} w^F_v}{\sum_{v \in \Gamma_F} \frac{w^F_v}{p^F_v}} \tag{6.4}$$

With $\Gamma_F = \{v \in F | E_{v=0} \neq 0 \wedge E_{v=1} \neq 0\}$ and $w^F_v = \frac{1}{|\Gamma_F|}$. We reject H_0 if we find $\mathring{p}_F \leq \alpha$.

One may wish to ignore a variable v if $E_{v=1}$ or $E_{v=0}$ is very small, as this will increase the required sample size. The reason is that the approximation to the χ^2 distribution in the GOF test is known to break down if any expected number of occurrences is below five [Yat34]. Thus, if we want $E_{v=i} \geq 5$, then we need to adjust

N accordingly. If one variable v is true (resp. false) in most models in R_F, then N will increase. Therefore, we may wish to ignore some variables with very low or very high frequencies if the computational cost is of importance.

Because we will have to repeat the VF test by using multiple formulae as input, we ask if we may compute the HMP of HMPs. By inserting the HMP formula into itself, we find:

$$\begin{aligned} \mathring{p}_R &= \frac{\sum_{F \in R} w_F}{\sum_{F \in R} \frac{w_F}{\mathring{p}_F}} \\ &= \frac{\sum_{F \in R} w_F}{\sum_{F \in R} \frac{w_F}{\frac{\sum_{v \in \Gamma_F} w_v^F}{\sum_{v \in \Gamma_F} \frac{w_v^F}{p_v^F}}}} \end{aligned} \tag{6.5}$$

With w_F, the weight of a p-value obtained by performing the VF test on a specific formula F and R, the subset of formulae on which the HMP is computed. We define w_v^F as the weight of a single test within the VF test performed on variable v of formula F.

In our case, because we compute the HMP over all the individual tests, we know that $\forall \Gamma_F : (\sum_{v \in \Gamma_F} w_v^F = 1)$. We thus obtain:

$$\begin{aligned} \mathring{p}_R &= \frac{\sum_{F \in R} w_F}{\sum_{F \in R} \frac{w_F}{\frac{1}{\sum_{v \in \Gamma_F} \frac{w_v^F}{p_v^F}}}} \\ &= \frac{\sum_{F \in R} w_F \sum_{v \in \Gamma_F} w_v^F}{\sum_{F \in R} \sum_{v \in \Gamma_F} \frac{w_F w_v^F}{p_v^F}} \\ &= \frac{\sum_{F \in R, v \in \Gamma_F} w_F w_v^F}{\sum_{F \in R, v \in \Gamma_F} \frac{w_F w_v^F}{p_v^F}} \end{aligned} \tag{6.6}$$

We conclude that in our case, computing the HMP of HMPs is correct, as it is equivalent to computing the HMP of the original p-values.

Selected Features per Configuration (SFpC) Statistical Test

Heradio et al. [HFG+20] proposed a statistical test called the selected features per configuration test (SFpC).

The main idea of the test is that a formula F will have n_k models with exactly k variables set to true and the remaining $|Var(F)| - k$ variables set to false with n_k defined as follows

$$n_k = |\{m \in R_F|\ k = \sigma(m)\}| \tag{6.7}$$

$$\sigma(m) = |\{v \in Var(F) | v \in m\}| \tag{6.8}$$

n_k can be computed for every $0 \leq k \leq |Var(F)|$ with $k \in \mathbb{N}$. If we compute $\frac{n_k}{|R_F|}$ for every k, we obtain the expected discrete probability distribution, i.e., the distribution that should be mimicked by the sampler if the sampler is uniform.

Testing a sampler is performed as follows. Let $S \in {R_F}^N$ be the tuple containing the sampled models. The authors follow by defining $E_k = n_k \frac{N}{|R_F|}$, the expected number of models with exactly k variables set to true and $O_k = |\{m \in S|\ \sigma(m) = k\}|$, the number of observed models in the sample S with exactly k variables set to true.

We compute Pearson's χ^2 test statistic as follows:

$$\chi^2 = \sum_{k \in \Gamma} \frac{(O_k - E_k)^2}{E_k} \tag{6.9}$$

with $\Gamma = \{k \in \mathbb{N}|\ 0 \leq k \leq |Var(F)|\ \wedge E_k \neq 0\}$. A p-value p is derived by performing Pearson's one-sided χ^2 test (with $|\Gamma| - 1$ degrees of freedom). We reject H_0 if we find $p \leq \alpha$ with α, a predefined significance level.

The original SFpC test implementation [HFG+20] uses the Jensen-Shannon divergence [Lin91] and computes the χ^2 test statistic with the results from [GBC+02]. Our implementation differs from the original implementation in [HFG+20] (by using equation 6.9 to compute the χ^2 test statistic) because we could not run their code.

Modbit Statistical Test

Next, we detail a variant of the SFpC test. The SFpC test can be very effective, but it tends to require large sample sizes. One effective way of reducing the sample size is to reduce the number of categories.

The SFpC test defines one category for each value $0 \leq k \leq |\,Var(F)|$. To reduce the number of categories, we define a category for each value $k \pmod q$ with q an integer such that $2 \leq q \leq |\,Var(F)| + 1$. Our strategy for this test is the following. We redefine n_k for the modbit test as follows.

$$n_k = |\{m \in R_F|\ k \equiv \sigma(m) \pmod q\}| \tag{6.10}$$

$$\sigma(m) = |\{v \in Var(F) | v \in m\}| \tag{6.11}$$

The remainder of the test is performed similarly to the SFpC test.

Let $S \in {R_F}^N$ be the tuple containing the sampled models. We follow by defining $E_k = n_k \frac{N}{|R_F|}$, the expected number of models with exactly $k \pmod q$ variables set

to true and $O_k = |\{m \in S |\ k \equiv \sigma(m) \pmod q\}|$, the number of observed models in the sample S with exactly $k \pmod q$ variables set to true.

We compute Pearson's χ^2 test statistic as follows:

$$\chi^2 = \sum_{k \in \Gamma} \frac{(O_k - E_k)^2}{E_k} \tag{6.12}$$

with $\Gamma = \{k \in \mathbb{N} |\ 0 \leq k \leq |Var(F)| \ \wedge E_k \neq 0\}$. A p-value p is derived by performing Pearson's one-sided χ^2 test (with $|\Gamma| - 1$ degrees of freedom). We reject H_0 if we find $p \leq \alpha$ with α, a predefined significance level.

The main strength of this test is its scalability. Since the number of categories can vary greatly with the parameter q, we can expect the required sample size N for the test to be relatively small when q is small. However, the reliability of the test is also limited by the fact that the number of categories is small when q is small. The test is unable to detect the non-uniformity of a sampler if the sampler under test somehow generates a sample with the right O_i values. This may happen if a sampler was specifically designed to have this property. To truly assess the uniformity of a sampler, other tests and larger values of q should be used.

Birthday Problem Statistical Test

We finish by introducing a test inspired by the birthday paradox and adapted from an existing test on PRNGs [ONe18]. Although the test is not necessarily considered to be the most reliable within the PRNG community, it does have some advantages. First, the test is simple enough to adapt to URS, as the required knowledge is limited to the sample size and the total number of models for the input Boolean formula. Second, it shows us whether a sampler produced either too many or not enough duplicates during the sampling process. If too many duplicates are produced, then the sample may be of little value as the number of unique models is low. If too few duplicates have been produced, then the sample may not be uniform, but at least it likely explores a larger portion of the hyperspace. We derive below the statistical test from the classical birthday problem [BH19] to test the hypothesis that a sampler samples uniformly at random.

The statistics considered, noted R, is the number of repeated models, i.e., the number of sampled unordered pairs where both models are equal. The distribution of the statistic R under the null hypothesis (uniform random sampling) is a well-known result of the 'birthday problem'. The number of repeated pairs R has approximately a Poisson distribution: $R \dot{\sim} \text{Pois}(\lambda)$ with $\lambda = \frac{\binom{N}{2}}{|R_F|}$, N the number of sampled models [BH19, p. 179]. Empirically, we sample N models with the sampler under test to compute the observed number of repeated models noted r. We consider a two-sided test here, since our alternative hypothesis can imply a deviation from uniform sampling by the right (a higher number of repeated models) or by the

left (a lower number of repeated models). The p-value of our two-sided birthday problem statistical test can be computed as follows:

$$\begin{aligned} p &= 2\min\{\mathbb{P}(R{\geq}\, r \mid H_0),\, \mathbb{P}(R{\leq}\, r \mid H_0)\} \\ &= 2\min\{1 - F_{\text{Pois}(\lambda)}(r-1),\, F_{\text{Pois}(\lambda)}(r)\} \end{aligned} \tag{6.13}$$

with $F_{\text{Pois}(\lambda)}(r)$ the cumulative distribution function (CDF) of the Poisson distribution parameterised by λ. Then, the p-value of our birthday problem statistical test allows us to reject or not the null hypothesis.

6.3 Experimental Study

We define below our research questions and the general experimental settings common to all our experiments. The specific settings of each research question are detailed in Section 6.4.

6.3.1 Research Questions

Our study aims to answer the following research questions:

1. **RQ1: Which URS tools are uniform according to the different statistical tests?** We assess the uniformity of the current implementations of state-of-the-art sampling methods. To achieve this, we apply our proposed statistical tests over a set of formulae and report the results for each statistical test and sampler.

2. **RQ2: What is the execution time of each statistical test?** Statistical tests have differences in reliability, meaning that the statistical values they focus on differ, leading to distinct abilities to detect non-uniformity. We therefore measure the execution time of each test on all samplers and input formulae. We contrast these measurements with the ability of each test to conclude non-uniformity.

3. **RQ3: How do different formula datasets compare in non-uniformity detection ability?** We hypothesise that statistical test outcomes are strongly bound to the formulae used and that this can be a source of bias in uniformity test results. Specifically, we investigate to what extent synthetic formulae — which are simpler and faster to process — are useful to conclude about samplers' uniformity on real-world formulae.

6.3.2 Datasets

We use a large number of well-known and publicly available models in our study, which are of various complexities and are either feature models or general Boolean formulae.

Feature Model Benchmark Properties

In total, we use the feature models of 128 real-world configurable systems (Linux, eCos, toybox, JHipster, etc.) with varying sizes and complexity. We first rely on 117 feature models used in [KTM+17; KTS+18]. Most feature models contain between 1,221 and 1,266 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295, and the other nine have between 49,770 and 50,606 cross-tree constraints [KTM+17; KTS+18]. Second, we include 10 additional feature models used in [LGC+15] and not in [KTM+17; KTS+18]; they also contain a large number of features (e.g., more than 6,000). Third, we also add the JHipster feature model [Rai15; HNA+18] to the study, a realistic but relatively smaller feature model (45 variables, 26,000+ configurations). We later refer to these benchmarks as the feature model benchmark. Once put in conjunctive normal form, these instances typically contain between 1 and 15 thousand variables and up to 340 thousand clauses. The hardest of them, modelling the Linux kernel configuration, contains more than 6 thousand variables, and 340 thousand clauses, and is generally seen as a milestone in configurable system analysis.

General Boolean Formulae

In addition to these feature models, we use the industrial SAT formulae as used in [DLB+18]. Since these formulae are much smaller than the feature models we use (typically a few thousand clauses), they will provide a basis of results for statistical analysis, in case a solver cannot produce enough models on the harder formulae. We later refer to these benchmarks as the non-feature model benchmarks.

Both of these datasets, the feature model benchmark and the general Boolean formulae, have been collected by Plazar et al. [PAP+19].

Filtered Dataset Ω

Performing tests on samplers requires us to generate a large number of models. Thus, to limit the computation time, we combine both the feature model dataset and the general Boolean formulae. We then ran the UniGen3 sampler on all of these formulae and measured the time and memory UniGen3 required to sample 1000 models. We removed all the formulae that required more than 10 minutes or more than 400MB of memory, which left us with 195 formulae. In the following sections, we will refer to this dataset as the Ω dataset.

We chose UniGen3 as a reference sampler as it has theoretical guarantees and it is the slowest sampler on our list according to our experiments (if we exclude Smarch). Given how slow Smarch was in our experiments, relying on Smarch to filter the dataset would have left us with a smaller dataset.

Synthetic Formulae

To test the importance of a dataset, we also generated synthetic k-CNF formulae. To generate a k-CNF formula, we use the classical k-CNF model used by [MSL92]. The model generates clauses of the desired length k until the formula has the desired number of constraints. A clause is generated by selecting k unique variables from a predefined set of variables. Each variable is negated with probability $\frac{1}{2}$.

The first synthetic dataset is r30c90, which contains 300 satisfiable 3-CNF formulae consisting of 30 variables and 90 clauses. The second synthetic dataset is r30c114, which contains 300 satisfiable 3-CNF formulae consisting of 30 variables and 114 clauses.

We also generated a dataset named r30c150b1000, which contains 300 3-CNF formulae. The r30c150b1000 dataset was generated with the model proposed by [EO22]. Each formula is generated with 30 variables and 150 clauses. Then, for each formula, 1000 random variable assignments are generated (not necessarily models of the formula). For each generated variable assignment, we check that each clause is satisfied by the assignment; if not, we randomly negate a literal of the unsatisfied clause. Negating a single literal is enough to render the clause satisfied.

The datasets and the programs used to generate the datasets are available on our companion GitHub [Zey25c].

6.3.3 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.

6.3.4 Computation Budget

We performed each statistical test on each sampler and each dataset. We define a unit as one statistical test, one sampler, and one dataset. Each unit had two days on a full node of the HPC. If the test was not finished by then, it would be interrupted, and we would use the available results.

As we require large amounts of models (and because some samplers are designed to generate no duplicate models, even though some statistical tests require them), we decided to generate models in batches. This means that we repeatedly called the samplers asking for 1000 models until the required number of models was met. The sampling process had a timeout of 5 hours. This means that if sampling the required number of models would take longer than 5 hours, then the test is not performed on the input formula (but may be performed on other formulae within the unit). Additional results for batch sizes of 2000 and 4000 models are available on our companion GitHub [Zey25c].

6.3.5 Hyperparameters

Some samplers require hyperparameters to function, and almost all require a random seed to initialise their pseudorandom number generator. In the absence of consistent documentation specifying seed formats or ranges, we conservatively restrict seeds to 31-bit signed integers (from 0 to $2^{31}-1$), which ensures compatibility with older and simpler PRNGs that are limited to 32-bit or 31-bit seeds. A new seed is generated for each invocation using Python's random module.

STS requires a buffer size, which is the number of partial assignments kept in memory during the search. We set the buffer size to the same value as the batch size to maximise model space exploration. This is larger than the default value of 50. For CMSGen and UniGen3, we use the default hyperparameters provided by the authors. In the version used, UniGen3 was configured with $\kappa = 0.638$, $\epsilon = 0.8$, and $\delta = 0.2$. For CMSGen, the `fixedconfl` parameter was set to 100, specifying the number of allowed conflicts before a restart is triggered.

The expected frequencies (E_i) required by the statistical tests were computed by first compiling the input formula to d-DNNF. We then applied algorithms similar to those presented in [HFM+19] to calculate the expected frequencies. Since the goodness-of-fit (GOF) test is known to be unreliable when any expected frequency falls below five [Yat34], and because our analysis relies on the GOF test, we conservatively set the sample size N such that all expected frequencies E_i exceed ten. As a result, the value of N depends on the structure of the input Boolean formula.

6.4 Results

6.4.1 RQ1: Uniformity of Samplers

In this section, we present the general results regarding uniformity performed on our Ω dataset. Tables 6.1 and 6.2 report the results. For each test, we split them into two columns: the first column is the number of formulae on which we managed to perform the test, and the second is the p-value. A bold p-value indicates that the p-value is greater than our predefined significance level $\alpha = 0.01$ (and thus is uniform according to our test). The tests were not necessarily performed on all the formulae because the sampler may have crashed or the sampling process may have been too time-consuming and thus interrupted before it could terminate, incidentally cancelling the test.

We observe that half of the samplers fail the modbit test on the Ω dataset, even for a low threshold ($q = 2$), with the exception of KUS, SPUR, UniGen3, and BDDSampler. Only UniGen3 and BDDSampler pass the variable frequency (VF) test. This result is of particular interest because SPUR, which is used as a reference sampler in Barbarik [MPC20], is detected as non-uniform by our tests. These

findings are consistent with those reported by Heradio et al. [HFG+22], reinforcing concerns about SPUR's uniformity. Among all evaluated tools, UniGen3 appears to be the only sampler that consistently passes all tests, suggesting it is likely the only truly uniform sampler in our study.

The birthday test fails to detect the non-uniformity of Smarch, which is identified by all other tests. Moreover, Smarch contains an off-by-one error, as demonstrated by its behaviour in edge cases: it crashes when given a formula with only one model and consistently returns the same model when the formula has exactly two models. This highlights the limitations of the birthday test as a standalone measure. While it is a useful test, it appears less reliable than others. Therefore, its results should be interpreted in conjunction with stronger tests such as the VF and SFpC tests.

Our results suggest that the VF test is a reliable method for detecting non-uniformity. The SFpC test serves as a useful complement to confirm VF results. The modbit test also shows strong reliability, but only when applied with higher values of q ($q \geq 64$).

BDDSampler performs well in most tests but fails the SFpC test and the modbit test at higher values of q ($q \geq 64$), indicating potential weaknesses under more demanding conditions. Given that BDDSampler comes with theoretical guarantees and only fails the more stringent tests, we hypothesise that this is due to a suboptimal choice of PRNG. To test this hypothesis, we applied the BigCrush test suite from the TestU01 library [LS07], which is specifically designed to evaluate the quality of PRNGs. The PRNG used by BDDSampler consistently failed the `WeightDistrib`, `SumCollector`, and `HammingIndep` tests, indicating that this PRNG is likely unsuitable for high-quality randomness.

The results obtained for KUS are somewhat surprising, given that KUS provides theoretical guarantees of uniformity. Fortunately, KUS outputs the model count of the parsed d-DNNF during execution, which allows for verification. By comparing these model counts with those produced by D4, we observed discrepancies in several cases. This suggests that KUS may contain a bug in either its d-DNNF parsing or model counting procedure, which could explain the observed deviations from uniformity in our tests. We note that KUS passed the GOF test. This is somewhat surprising, as it contradicts our other results. However, given that the GOF test was performed successfully on a much lower number of formulae, we argue that the test is less reliable.

To further validate our general conclusion and mitigate potential biases introduced by the chosen batch size, we repeated our experiments with batch sizes of 2000 and 4000 models. The results are available in our companion GitHub repository [Zey25c]. We reached the same conclusion across different batch sizes, indicating that batch size has a negligible influence on our results.

As noted above, most samplers are not uniform. Moreover, it is well-known that

	VF		Birthday		SFpC		GOF	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value
KUS	192	0.000	142	0.001	92	0.000	69	**0.138**
QuickSampler	186	0.000	139	0.000	77	0.000	62	0.000
Smarch	127	0.000	78	**0.023**	24	0.000	28	0.000
SPUR	189	0.000	145	0.000	99	0.000	78	0.000
STS	191	0.000	138	0.000	81	0.000	69	0.000
CMSGen	143	0.000	93	0.000	71	0.000	61	0.000
UniGen3	183	**0.083**	130	**0.274**	76	**0.253**	71	**0.353**
BDDSampler	118	**0.099**	92	**0.274**	68	0.000	66	0.000

Table 6.1: Experimental results for the Ω dataset. 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).

	q = 2		q = 8		q = 32		q = 64	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value
KUS	193	**0.137**	193	0.000	189	0.000	135	0.000
QuickSampler	189	0.000	188	0.000	188	0.000	126	0.000
Smarch	187	0.000	186	0.000	142	0.000	46	0.000
SPUR	193	**0.142**	193	**0.185**	193	0.000	142	0.000
STS	193	0.000	193	0.000	192	0.000	132	0.000
CMSGen	144	0.000	144	0.000	144	0.000	116	0.000
UniGen3	192	**0.159**	192	**0.234**	192	**0.268**	122	**0.116**
BDDSampler	118	**0.165**	118	**0.182**	118	**0.058**	76	0.000

Table 6.2: Experimental results for modbit test on the Ω dataset. 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).

	Uniformity		Observed number of repetitions			
Sampler	#F	p-value	min	max	average	median
KUS	142	0.001	0	18	9.01	10
QuickSampler	139	0.000	0	29858	480.01	4
Smarch	78	**0.023**	3	22	10.71	10
SPUR	145	0.000	4	307	34.48	12
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
BDDSampler	92	**0.274**	3	17	9.96	10

Table 6.3: Extended experimental results for the birthday test with the Ω dataset. 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).

there exist formulae for which uniform sampling does not scale. Thus, heuristic-based samplers become interesting. However, not all heuristic-based samplers are equal.

The birthday test provides very insightful information on samplers (in addition to the p-value), as the birthday test studies the number of repetitions. The observed number of repetitions may be of particular interest if the user wishes to use a heuristic-based sampler. A sampler with too many repetitions is non-uniform and often returns the same model. A sampler with too few repetitions is non-uniform and seldom returns the same model. The first case would lead to the exploration of a smaller part of the model space. On the other hand, generating fewer repetitions than necessary for uniformity on large model spaces would indicate a better exploration of said model space. Suppose that a user wishes to sample uniformly from a formula, but none of the theoretically uniform tools fit into the user's budget. The user thus has the choice between QuickSampler, CMSGen, and STS. The user may sample from the target formula by using all the samplers and comparing the observed numbers of repetitions. The sampler generating the fewest repetitions is likely a good candidate.

Table 6.3 shows the detailed results for the Birthday test on our Ω dataset. In our experiments, we set the expected number of repetitions to be approximately 10, as shown by the average number of repetitions observed on the models generated by UniGen3 (9.78). The CMSGen sampler generated on average 991.37 repetitions. This number is likely too high for most users. The same is true for QuickSampler, which generated an average of 480.01 repetitions. STS on the other hand generated

	VF		Birthday		SFpC		GOF	
Sampler	#F	time (h)	#F	time (h)	#F	time (h)	#F	time (h)
KUS	192	7.3	142	12.3	92	52.8	69	33.4
QuickSampler	186	17.3	139	32.3	77	63.1	62	42.2
Smarch	127	143.8	78	69.1	24	14.2	28	21.7
SPUR	189	4.4	145	19.9	99	37.6	78	14.4
STS	191	26.7	138	20.8	81	38.1	69	26.0
CMSGen	143	1.1	93	16.5	71	16.8	61	30.3
UniGen3	183	31.2	130	51.1	76	41.0	71	32.4
BDDSampler	118	3.2	92	4.5	68	9.6	66	8.6

Table 6.4: Scalability results for the Ω dataset. For each test (and for each formula), each sampler was called multiple times to generate samples of size 1000. The indicated time (in hours) is the accumulated time across all the formulae for which the test was performed successfully. #F indicates the number of formulae on which the test was successfully performed (i.e., without timeouts or out-of-memory errors).

an average of 5.2 repetitions, which is too low to be uniform but likely preferable to QuickSampler and CMSGen. Thus, a user who cannot afford uniform sampling should likely rely on STS for their sampling tasks.

Answer to RQ1: According to our results, the modbit test is not very reliable for lower values of q ($q < 64$). The variable frequency test detected the non-uniformity of every sampler, except for UniGen3 and BDDSampler. The SFpC and modbit ($q \geq 64$) tests detected a lack of uniformity in the sample generated with BDDSampler. The GOF test requires too many models to be usable, given the computational cost of sampling.

The birthday test is an interesting addition to the set of tests. It allows for uniformity testing and provides further insight into the capacity of samplers to explore the model space.

6.4.2 RQ2: Scalability

In this subsection, we explore the scalability of each test, or in other words, the computational cost of each statistical test presented in this chapter.

Tables 6.4 and 6.5 show the sequential execution time of each test for the Ω dataset on each sampler. The sequential execution time is the sum of the execution times for each formula computed on the indicated number of formulae (and it does not contain the formulae that timed out or had an error). A first striking observation is that Smarch is significantly slower than all the other samplers. A

	q = 2		q = 8		q = 32		q = 64	
Sampler	#F	time (h)	#F	time (h)	#F	time (h)	#F	time (h)
KUS	193	0.8	193	0.8	189	5.1	135	34.9
QuickSampler	189	6.6	188	4.0	188	11.2	126	61.9
Smarch	187	205.6	186	203.0	142	218.7	46	52.6
SPUR	193	4.1	193	4.2	193	5.5	142	38.0
STS	193	8.1	193	8.1	192	23.1	132	47.7
CMSGen	144	0.1	144	0.1	144	0.8	116	26.4
UniGen3	192	2.5	192	2.6	192	11.2	122	58.1
BDDSampler	118	0.7	118	0.7	118	6.8	76	7.6

Table 6.5: Scalability results for the modbit test on the Ω dataset. For each test (and for each formula), each sampler was called multiple times to generate samples of size 1000. The indicated time (in hours) is the accumulated time across all the formulae for which the test was performed successfully. #F indicates the number of formulae on which the test was successfully performed (i.e., without timeouts or out-of-memory errors).

second, more subtle observation is that the execution time increases when switching from the modbit ($q \leq 32$) test to the VF test and from the VF test to the SFpC test. At the same time, the number of formulae on which the tests were performed successfully gradually decreases from one test to another. This indicates that the SFpC test requires more computation time than the VF test, which requires more computation time than the modbit ($q \leq 32$) test. The birthday test is more nuanced regarding the accumulated computation time. However, regarding the number of formulae on which the birthday test was performed successfully, we find that the birthday test is more expensive than the VF test and cheaper than the SFpC test. The GOF test was successfully performed on the smallest number of formulae. We thus conclude that GOF is the most expensive test to perform.

Given our results, if we order the tests according to their computational cost, we find the following ordering: Modbit ($q \leq 32$) $\leq_C$ VF $\leq_C$ Birthday $\leq_C$ Modbit ($q \geq 64$) $\leq_C$ SFpC $\leq_C$ GOF, with modbit ($q \leq 32$) the cheapest test and GOF the most expensive test.

Smarch seems to observe a decrease in execution time when reading the table according to our ordering. However, this is coupled with a drastic decrease in the number of formulae successfully processed.

By considering our results on scalability and uniformity, we propose the ordering presented in Figure 6.1. The dashed lines indicate optional transitions. A user with computational budget limitations would start testing with the VF test and then proceed with the SFpC test. If the user has a bigger computational budget, then they

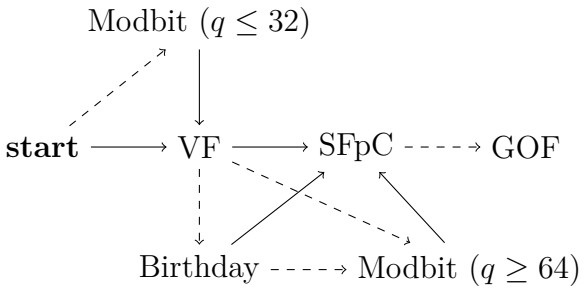


Figure 6.1: Statistical test ordering for uniform random sampling. The dashed lines indicate optional transitions. A user may thus adapt the executed tests depending on their needs and computational budget.

may include the modbit ($q \leq 32$) test before the VF test or the birthday and modbit ($q \geq 64$) tests after the VF test. The choice depends on the user's requirements. If the user wants to analyse the observed number of repetitions, then the birthday test is interesting. However, if the user has many non-uniform samplers, then the modbit ($q \leq 32$) test may be a good candidate for fast 'pre-testing'. Additionally, the choice depends on the user's computational budget, as the modbit ($q \leq 32$) test is cheaper than the birthday test. The GOF test is optional as it only scales to the smallest formulae.

Answer to RQ2: After an analysis of the computational cost of each test, we find the following ordering: Modbit ($q \leq 32$) $\leq_C$ VF $\leq_C$ Birthday $\leq_C$ Modbit ($q \geq 64$) $\leq_C$ SFpC $\leq_C$ GOF, with Modbit ($q \leq 32$) the cheapest test and GOF the most expensive test.

We recommend performing the tests according to the ordering in Figure 6.1. If performing every test is too expensive, we recommend excluding the GOF test first.

6.4.3 RQ3: On the Influence of Formula Choice

In this section, we explore the influence of the dataset on the uniformity results. The goal is to understand if the dataset influences the uniformity result returned by our statistical tests. A negative answer would allow us to focus our testing on small formulae that are easy to sample from. A positive answer would mean that to obtain reliable results, one needs to choose their dataset carefully.

To explore the influence of the dataset, we also performed the statistical tests with synthetic formulae. We may then compare these results with the results on real-world formulae.

	Modbit q = 32		VF		Birthday		SFpC		GOF	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value	#F	p-value
KUS	300	**0.133**	300	**0.065**	300	**0.061**	300	**0.131**	300	**0.096**
QuickSampler	300	0.000	300	0.000	300	0.000	300	0.000	300	0.000
Smarch	39	**0.074**	293	**0.017**	300	**0.201**	42	**0.134**	23	**0.036**
SPUR	300	**0.105**	300	**0.106**	300	0.000	300	**0.129**	300	0.000
STS	300	**0.069**	300	0.000	300	0.000	300	0.007	300	**0.993**
CMSGen	300	0.000	300	0.000	300	0.000	300	0.000	300	0.000
UniGen3	300	**0.100**	300	**0.102**	300	**0.207**	300	**0.028**	300	**0.151**
BDDSampler	300	**0.120**	300	**0.093**	300	**0.100**	300	**0.134**	300	**0.169**

Table 6.6: Experimental results for the r30c90 dataset. 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).

Uniformity

We start by exploring the uniformity results obtained by performing the statistical tests on synthetic datasets. Tables 6.6, 6.7, and 6.8 show the uniformity results on the synthetic datasets. As with Table 6.1, a bold p-value indicates that the p-value is greater than our predefined significance level $\alpha = 0.01$ (and is thus uniform according to our test).

Our first observation is that KUS seems to be uniform on the synthetic formulae as it fails no test. We also observe that CMSGen fails the modbit test on every synthetic dataset. Moreover, CMSGen fails every other test on every dataset, and we thus conclude that CMSGen is not uniform.

We follow by observing that the dataset used in Table 6.6 is slightly more effective at detecting non-uniformity than the dataset used in Table 6.7. However, the VF test fails to detect the non-uniformity of Smarch with the r30c90 dataset, while it succeeds with the r30c114 dataset. Thus, to detect the non-uniformity of as many samplers as possible by using the VF test and synthetic datasets, we would need to combine the results obtained with both datasets, r30c90 and r30c114. The dataset used in Table 6.8 seems to add little value to the other synthetic datasets.

We continue this subsection by comparing the uniformity results with the ones found in Table 6.1, which were computed on our Ω dataset. This should give us insights into the importance of dataset choice.

We start by noting that overall, the tests are significantly less reliable on synthetic formulae as they eliminate fewer samplers than in Table 6.1. Next, our results on synthetic formulae indicate that KUS is uniform and gives us mixed results on STS. The results in Table 6.1 are much clearer as both STS and KUS fail every test

	Modbit q = 32		VF		Birthday		SFpC		GOF	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value	#F	p-value
KUS	300	**0.205**	300	**0.053**	300	**0.127**	300	**0.177**	300	**0.161**
QuickSampler	296	0.000	296	0.000	300	0.000	296	0.000	300	0.000
Smarch	288	0.000	296	0.000	296	**0.061**	288	0.000	284	0.001
SPUR	300	**0.101**	300	**0.084**	300	0.000	300	**0.216**	300	0.000
STS	300	**0.903**	300	**0.077**	300	0.000	300	**0.986**	300	**1.000**
CMSGen	300	0.000	300	0.000	300	0.000	300	0.000	300	0.000
UniGen3	300	**0.130**	300	**0.109**	300	**0.110**	300	**0.175**	300	**0.062**
BDDSampler	300	**0.161**	300	**0.076**	300	**0.180**	300	**0.138**	300	**0.164**

Table 6.7: Experimental results for the r30c114 dataset. 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).

	Modbit q = 32		VF		Birthday		SFpC		GOF	
Sampler	#F	p-value	#F	p-value	#F	p-value	#F	p-value	#F	p-value
KUS	300	**0.197**	300	**0.097**	300	**0.169**	300	**0.077**	300	**0.188**
QuickSampler	300	0.000	300	0.000	300	0.000	300	0.000	300	0.000
Smarch	43	**0.158**	281	**0.100**	300	**0.010**	49	**0.198**	26	**0.124**
SPUR	300	**0.061**	300	**0.057**	300	0.000	300	**0.170**	300	0.000
STS	300	0.000	300	0.000	300	0.000	300	0.008	300	**1.000**
CMSGen	300	0.000	300	0.000	300	0.000	300	0.000	300	0.000
UniGen3	300	**0.074**	300	**0.065**	300	**0.104**	300	**0.124**	300	**0.032**
BDDSampler	300	**0.161**	300	**0.087**	300	**0.178**	300	**0.087**	300	**0.157**

Table 6.8: Experimental results for the r30c150b1000 dataset. 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).

Sampler	Modbit q = 32		VF		Birthday		SFpC		GOF	
	#F	time (h)	#F	time (h)	#F	time (h)	#F	time (h)	#F	time (h)
KUS	300	0.8	300	0.0	300	0.0	300	0.8	300	1.1
QuickSampler	300	2.5	300	0.2	300	0.1	300	2.5	300	3.5
Smarch	39	100.2	293	146.0	300	118.7	42	107.0	23	55.4
SPUR	300	0.3	300	0.0	300	0.0	300	0.3	300	0.4
STS	300	0.3	300	0.0	300	0.0	300	0.3	300	0.4
CMSGen	300	0.1	300	0.0	300	0.0	300	0.1	300	0.2
UniGen3	300	1.5	300	0.1	300	0.0	300	1.5	300	2.3
BDDSampler	300	0.1	300	0.0	300	0.0	300	0.1	300	0.2

Table 6.9: Scalability results for the r30c90 dataset. For each test (and for each formula), each sampler was called multiple times to generate samples of size 1000. The indicated time (in hours) is the accumulated time across all the formulae for which the test was performed successfully. #F indicates the number of formulae on which the test was successfully performed (i.e., without timeouts or out-of-memory errors).

except for the GOF test (and the modbit ($q < 8$) test for KUS). Thus, we conclude that the choice of the dataset is important for reliable test results.

Scalability

We continue this subsection by exploring the difference in scalability between the real-world dataset and the synthetic datasets. This will give us further insights regarding the relevance of synthetic datasets.

Table 6.9 shows the sequential execution time of each test for the r30c90 dataset on each sampler. We observe that except for the Smarch sampler, the execution times are generally very low (under 1 hour).

By comparing Tables 6.4 and 6.9, we notice that the synthetic dataset is significantly faster to process than the Ω dataset. As an example, the VF test on STS required 26.7 hours to be performed on the Ω dataset and required around 0.02 hours to be performed on the r30c90 dataset. The SFpC time budget for STS on the synthetic dataset was 0.3 hours, which is also less than required by the VF test on the Ω dataset and also less than the modbit ($q \leq 16$) test on the Ω dataset (8.3 hours). However, all of these tests have the same conclusion: STS is not uniform. We thus argue that fast 'pre-testing' may be performed by using synthetic datasets to detect the majority of non-uniform samplers quickly. This would allow one to filter a lot of non-uniform samplers while testing and only perform more expensive tests on more expensive datasets if a sampler 'survives' the 'pre-testing'.

Answer to RQ3: We conclude that the dataset choice can drastically bias the uniformity conclusion. Specifically, if a sampler fails a test on a synthetic

benchmark, then it is unlikely to pass the test on a real-world formula. The contraposition, however, does not hold. Thus, we recommend using synthetic formulae to quickly eliminate a large number of samplers before considering larger, real-world formulae.

6.4.4 Discussion on Uniformity and Statistical Test Results

Every statistical test presented in this article has different strengths and weaknesses. For example, the modbit test operates on q categories. It is thus likely to require fewer samples for small values of q as this means that it is easier to have the required number of observations in each category (unless there is a strong imbalance between the q categories). The low sample size is the modbit test's biggest strength. However, if the non-uniformity of a sampler does not generate an effect across these q categories, then the modbit test will not detect it, thus the test's biggest weakness. A sampler may be engineered to specifically pass the modbit test. Our results show that SPUR is uniform according to the modbit ($q < 32$) test but not uniform according to the VF test, which further confirms our claim that using a single statistical test gives unreliable results. This is also the conclusion of the PRNG community as it is standard practice in statistical testing of pseudo-random number generators (PRNG) [RSN+01].

We observe from our results that formula choice has an impact on test results. For example, KUS was detected as not uniform on the real-world formulae while being uniform on the synthetic datasets. This gives us two insights. First, testing a sampler on a single formula is not enough. Second, the used dataset should contain a variety of formulae to maximise the coverage of a sampler (similar to software testing).

Our next observation is that in most cases, a sampler (whether uniform or not) will usually fail a test on at least a few formulae. Statistical tests have a non-zero probability of returning the wrong answer (i.e., Type I error). That is, sometimes a non-uniform sampler will pass a test, and sometimes a uniform sampler will fail a test. If a sampler is uniform, then the p-values returned by the statistical tests will also have a uniform distribution [KPN09]. This means that a sampler failing a test on a single formula (if we test with multiple formulae) is not enough to determine if the sampler is not uniform. This raises the question: how many failed tests are too many? Luckily, the issue of performing multiple statistical tests and its consequences has been studied. We use the HMP to mitigate these issues. By using the HMP, we can summarise the test results for a dataset with a single p-value, thus simplifying the interpretation.

To conclude, we need to test our samplers by using multiple tests and by using a diverse set of formulae to explore the uniformity of the samplers as much as possible.

Performing multiple tests means that we may find a sampler that will pass some tests and fail others. To decide whether a sampler is uniform or not, we look at the consistency of the results instead of the specific results. If we test a sampler with five tests and the sampler fails three of the tests, then we may reject the uniformity of the sampler with high confidence. However, if a sampler only fails one test, then the sampler is likely uniform, and the failed test may be due to bad luck.

6.5 Threats to Validity

As for any empirical study, there are some threats to consider.

Internal Validity

As with any statistical test, there is the possibility of wrongly rejecting or accepting H_0. We mitigate the probability of falsely rejecting H_0 by choosing a low significance level $\alpha = 0.01$. We believe that falsely accepting H_0 is not an issue, considering our results (UniGen3 being the only sampler that is deemed uniform by our tests and having theoretical guarantees of uniformity).

External Validity

We cannot guarantee that our findings generalise to any formula and every sampler. There are multiple reasons behind this. For example, a sampler could be specifically engineered to overfit our battery of statistical tests without being uniform, which would require the development of new statistical tests. Another example was presented in this chapter with the synthetic formula benchmarks. As shown in our results, the quality of the dataset is important to have reliable results. As such, we cannot guarantee that there does not exist a dataset A, such that a sampler would be uniform on the real-world formulae used in this chapter but not uniform on the dataset A. To mitigate these threats, we selected multiple statistical tests that test different properties of the sampler and a range of SAT formulae from multiple sources. The formulae encode different types of models: Electronic circuits, algorithmic problems, Linux kernels, Unix command line tools, configuration tools, etc. [CMV14; CFM+15; DLB+18; PAP+19; APC21; HNA+18].

6.6 Conclusion

To conclude, we developed a series of statistical tests to test the practical uniformity of uniform random samplers. By using these tests, we have shown that most sampler implementations do not produce samples with uniform distribution, thus demonstrating the need for more systematic testing of uniform random samplers in general. Systematic testing would allow sampler developers to catch bugs in their implementation (if the sampler is not heuristic-based) before publication. In addition, our battery of tests can be used to accelerate the development of new samplers as theoretical proofs of uniformity become less important.

Furthermore, we used synthetic formulae to demonstrate that the dataset used to test the uniform random samplers is important. We have shown that before any testing is done, a dataset should be constructed from a wide range of sources to achieve reliable results. Sadly, these results also imply that exclusively using small formulae to lower the computational cost of testing is not a good idea. If a sampler is deemed non-uniform on a computationally easy dataset, then further testing is likely not necessary. If, however, the sampler is deemed uniform, then further testing on more diverse and often more computationally demanding datasets is likely necessary.

Finally, we would like to highlight that all our results are available on our companion GitHub [Zey25c]. Adding samplers and statistical tests to the repository is possible via pull requests, thus creating a common playground for future statistical tests and uniform random samplers.

7

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

At the time of writing, knowledge compilation is one of the most effective ways to achieve uniform random sampling, as shown by D4 and KUS. Nevertheless, some formulae remain out of reach. In this chapter, we present a novel divide-and-conquer approach to knowledge compilation.

Contents

7.1 Introduction

Knowledge compilation (KC) is the problem of transforming Boolean formulae into alternative representations that allow for more efficient reasoning [DM02]. 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. KC can mitigate this intractability by transforming CNF formulae into target languages amenable to #SAT [SRH+24] and URS [SGR+18].

One such language is the deterministic decomposable negation normal form (d-DNNF) [Dar00], which is known to scale well in practice [SHN+23]. Efficient compilers for the d-DNNF language exist — most prominently D4 [LM17] — thereby enabling efficient solving of #SAT and URS. Despite these significant advances, formulae with large and intricate model spaces remain out of reach for existing compilers [SHN+23]. Approximate algorithms exist for #SAT like ApproxMC 7 [PMY25]. 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 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 114 formulae to d-DNNF out of the 672 formulae that were previously out of reach for D4 [LM17]. 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 (9.5×10^{-9} 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 in Chapter 6. All

of the programs and experimental results are available on our companion GitHub [Zey25a].

7.2 DivKC

Our approach is based on the idea that splitting a formula F into smaller parts will make it easier to compile the formula into a target language. More specifically, in Chapter 4 we 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.

7.2.1 Overview of the Decomposition Algorithm

Algorithm 2 *Compile*(F)

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

A high-level description of our approach is shown in Algorithm 2. To compile a formula F into d-DNNF, we begin by applying a function $\mathit{Split}(F)$, which returns a set of variables $P \subseteq \mathit{Var}(F)$ that will be used for projection (we present the algorithm to appropriately determine this subset P in Section 7.2.2). 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 [SM21; LLM16]. 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 $m \in R_\Gamma$. Let $G'_U = F \setminus G_U = \{c \in F | \, Var(c) \subseteq P\}$. Hence, $R_F = R_{G_U} \cap R_{G'_U}$. Therefore, to prove $m \in R_F$ we need to prove $m \in R_{G'_U}$ and $m \in R_{G_U}$. By definition of Γ we have $m \in R_{G_U}$. We continue by proving $m \in R_{G'_U}$. 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 $y \subseteq m$. 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 [DM02].

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. □

7.2.2 Choosing the Projection Set P

We decided to use hypergraph partitioning to choose a good projection set P. Other methods exist [AGL12], but hypergraph partitioning offers a good balance between simplicity and efficiency. While hypergraph partitioning is a very difficult problem to solve, efficient solvers do exist [ÇA11]. Our approach is described in Algorithm 3.

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 $v \in Var(F)$. 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 $v \in Var(c)$.

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 $Var(F)$ 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 $\Delta = \{c \in F | \exists x, y \in c : (p(\mathit{Var}(x)) \neq p(\mathit{Var}(y)))\}$. 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 3 *Split*(F)

Require: F is a Boolean formula in CNF
1: $vig \leftarrow \{\mathit{Var}(c) | c \in F\}$
2: $p \leftarrow$ *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: $P \leftarrow P \cup \{\mathit{Var}(l) | l \in c\}$
8: **end if**
9: **end for**
10: $\{P$ contains the variables of every clause that connects multiple partitions.$\}$
11: **return** P

7.2.3 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 7.1. We consider $F = (a \vee b) \wedge (c \vee d) \wedge (a \vee c)$. Selecting $P = \{a, c\}$ yields $G_U = (a \vee b) \wedge (c \vee d)$ and $G_P = a \vee c$. 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 7.1. 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.

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

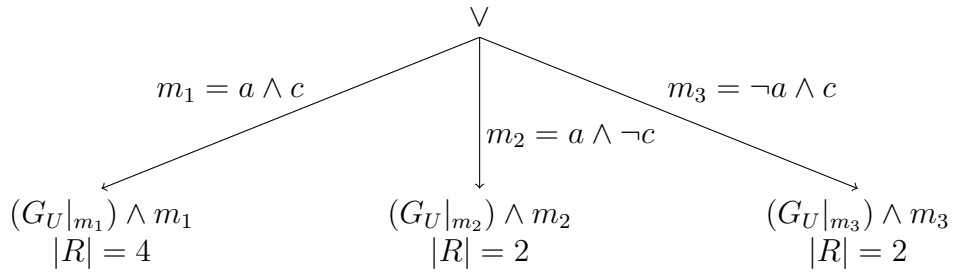


Figure 7.1: The d-DNNF that we obtain by using Algorithm 2 to compile $F = (a \vee b) \wedge (c \vee d) \wedge (a \vee c)$ with $P = \{a, c\}$.

shown by Lagniez and Lonca [LL24], 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 $G'_U = \{c \in F | \, Var(c) \subseteq P\}$. 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 $|a| < |\, Var(G_P)|$ 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 generates orthogonal sufficient partial assignments. In practice, we can focus on the number of paths that start at the root of the directed acyclic graph that represents the d-DNNF of G_P instead of the number of models $|R_{G_P}|$, with each path representing a sufficient partial assignment to the variables of G_P and with the d-DNNF requirements ensuring orthogonality (i.e., there does not exist a model y that is a superset of more than one path in the d-DNNF) [LL24].

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 4 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 $|R_F| = \sum_{a \in R_{G_P}} |R_{(G_U|_a) \wedge a}|$. 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 4), provides a consistent unbiased estimator of $|R_F|$ as a consequence of the law of large numbers.

Moreover, we use the central limit theorem 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 $1 - \alpha$.

Algorithm 4 *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

7.2.4 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 5.

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.

Lemma 2. *The probability $P(m)$ that Algorithm 5 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) numbers 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: $P_H \leq P(m|S) \leq P_L$, 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 $k \to |R_{G_P}|$.

7.3 Experimental Evaluation

We report on experiments assessing the benefits of DivKC. First, we demonstrate the ability of Algorithm 2 to compile into the d-DNNF form for challenging formulae that D4 [LM17] 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 4 in terms of coverage and precision. For coverage, we measure the percentage of formulae for which the true model count lies within the interval. 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.

Algorithm 5 *K-Sampler*(G_P, G_U, k, N)

1: $R \leftarrow \emptyset$
2: **for all** $1 \leq i \leq N$ **do**
3: $S \leftarrow$ *subset of size k from* R_{G_P}
4: $L \leftarrow \emptyset$
5: $lmc \leftarrow 0$
6: **for all** $a \in S$ **do**
7: $s_i \leftarrow |R_{(G_U|_a) \wedge a}|$
8: $L \leftarrow L \cup \{(a, s_i)\}$
9: $lmc \leftarrow lmc + s_i$
10: **end for**
11: $id \leftarrow$ *random_number*$(1 \leq r \leq lmc)$
12: **for all** $(a, s_i) \in L$ **do**
13: **if** $id \leq s_i$ **then**
14: $R \leftarrow R \cup \{$*random_model*$((G_U|_a) \wedge a)\}$
15: break current loop
16: **else**
17: $id \leftarrow id - s_i$
18: **end if**
19: **end for**
20: **end for**
21: **return** R

Third, we evaluate the ability of DivKC to generate uniform random samples for various formulae. For this, we generate samples using Algorithm 5 and execute the uniformity test suite proposed in Chapter 6.

7.3.1 Experimental Setup

Datasets

To test the compilation and model counting abilities of DivKC, we collected multiple datasets from Lagniez and Marquis [LM17], Soos [Soo24], Sundermann et al. [SBK+24], and Plazar et al. [PAP+19]. For the sake of fine-grained traceability, we keep these datasets and their identified subsets separated in our result table. Table 7.1 presents the characteristics of the formulae included in these (sub)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 [LM17] and Plazar et al. [PAP+19] datasets are further broken down into smaller sub-datasets, as each sub-dataset may come from different sources and may thus have different properties.

The Lagniez and Marquis [LM17] 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 [Soo24] dataset contains 1896 formulae from various sources, including the Model Counting Competitions.

The Sundermann et al. [SBK+24] 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. [PAP+19] dataset contains 503 formulae consisting of a feature model benchmark as well as other formulae collected from [DLB+18].

To test the uniformity of our approach, we used the same dataset as in Chapter 6 to allow for a direct comparison with our previous results. Using the same dataset allows us to use the results in Chapter 6 as a baseline and use our results to construct a direct comparison between the samplers tested in Chapter 6 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 [LM17] is limited to 64GB of memory and five hours of

Dataset	$\#F_{total}$	min(|$Var(F)$|)	max(|$Var(F)$|)	min(|F|)	max(|F|)
Global	4656	0	8286433	0	7689680
↪ Lagniez [LM17]	1979	5	229100	10	399794
↪↪ Bayesian Network	1116	32	229100	38	399794
↪↪ BMC	18	762	63624	2469	368352
↪↪ Circuit	68	26	8704	66	83902
↪↪ Configuration	35	1400	2038	1698	11342
↪↪ Handmade	68	61	3176	254	174160
↪↪ Planning	557	5	24816	10	148891
↪↪ QIF	7	251	4473	452	14011
↪↪ Random	104	75	150	150	525
↪↪ Scheduling	6	19500	22500	103887	123329
↪ Soos [Soo24]	1896	2	8286433	1	7689680
↪ Plazar [PAP+19]	503	14	486193	31	2598178
↪↪ Blasted Real	210	14	10329	35	33008
↪↪ Feature Models	133	45	62482	104	343944
↪↪ V15	30	17	25615	43	57946
↪↪ V3	30	17	25528	31	57586
↪↪ V7	30	17	25546	35	57662
↪↪ unsorted	70	67	486193	66	2598178
↪ Sundermann [SBK+24]	278	0	31012	0	350120

Table 7.1: 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.

computation for each formula. Algorithm 2 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 [LM17]. Therefore, Algorithm 2 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 4) 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 [LM17].

7.3.2 Experimental Results

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 7.2. The $\#F_{total}$ column indicates the total number of formulae in each dataset. The main columns for our evaluation are #¬D4 and #DivKC ∧ ¬D4. The former shows how

Dataset	$\#F_{total}$	#D4 ∧ ¬DivKC	#¬D4	#DivKC ∧ ¬D4
Global	4656	643	672	**114**
↪ Lagniez [LM17]	1979	256	214	**54**
↪↪ Bayesian Network	1116	127	70	**53**
↪↪ BMC	18	12	2	0
↪↪ Circuit	68	2	8	0
↪↪ Configuration	35	11	1	0
↪↪ Handmade	68	1	32	0
↪↪ Planning	557	102	92	**1**
↪↪ QIF	7	1	1	0
↪↪ Random	104	0	2	0
↪↪ Scheduling	6	0	6	0
↪ Soos [Soo24]	1896	311	394	**54**
↪ Plazar [PAP+19]	503	58	61	**6**
↪↪ Blasted Real	210	7	30	**3**
↪↪ Feature Models	133	13	5	0
↪↪ V15	30	2	5	**1**
↪↪ V3	30	2	5	**1**
↪↪ V7	30	2	5	**1**
↪↪ unsorted	70	32	11	0
↪ Sundermann [SBK+24]	278	18	3	0

Table 7.2: Experimental results regarding the scalability of Algorithm 2. Column $\#F_{total}$ indicates the total number of formulae in each dataset. The next column shows the number of formulae compiled only by D4 [LM17] but not by Algorithm 2. Column #¬D4 shows the number of formulae not compiled by D4. The last column indicates the number of formulae that were only compiled by Algorithm 2, but not by D4.

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 is that out of the 672 formulae that D4 could not initially compile (within our computational budget and with our current hardware), 114 of them can be compiled by our DivKC approach (using D4 as a backbone d-DNNF compiler).

Approximate Model Counting

We measure the coverage (Table 7.3) and precision (Table 7.4) of the intervals constructed by Algorithm 4 executed with parameters $\alpha = 0.01$ and $N = 10{,}000$. We constrained the computational budget to 64GB of memory and a maximum

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 [LM17]	1509	0.993	0.914	0.907	1.000
↪↪ Bayesian Network	919	0.992	0.905	0.898	1.000
↪↪ BMC	4	1.000	1.000	1.000	1.000
↪↪ Circuit	58	1.000	0.862	0.862	1.000
↪↪ Configuration	23	1.000	0.696	0.696	1.000
↪↪ Handmade	35	1.000	1.000	1.000	1.000
↪↪ Planning	363	0.989	0.934	0.923	1.000
↪↪ QIF	5	1.000	1.000	1.000	1.000
↪↪ Random	102	1.000	0.961	0.961	1.000
↪↪ Scheduling	0				
↪ Soos [Soo24]	1191	0.981	0.915	0.896	1.000
↪ Plazar [PAP+19]	384	0.987	0.815	0.802	1.000
↪↪ Blasted Real	173	0.994	0.971	0.965	1.000
↪↪ Feature Models	115	1.000	0.478	0.478	1.000
↪↪ V15	23	1.000	1.000	1.000	1.000
↪↪ V3	23	1.000	1.000	1.000	1.000
↪↪ V7	23	1.000	0.957	0.957	1.000
↪↪ unsorted	27	0.852	0.815	0.667	1.000
↪ Sundermann [SBK+24]	257	0.996	0.475	0.471	1.000

Table 7.3: Experimental results for Algorithm 4. Column #F indicates with how many formulae the following statistics have been computed. Column $Y_l \leq |R_F|$ indicates how often the lower bound returned by Algorithm 4 is correct (i.e., smaller than the true model count of F). Similarly, column $Y_h \geq |R_F|$ indicates how often the upper bound is correct. 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 confirms the correctness of the bounds obtained using Lemma 1.

runtime of 1.5 hours per execution of Algorithm 4.

In Table 7.3, the column #F shows for how many formulae we managed to compute both the exact model count with D4 [LM17] (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.

Concerning Algorithm 4, 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. [PAP+19] dataset and in 99% of the cases for the Sundermann et al. [SBK+24] dataset. However, the upper bound is only correct in 47% of the cases. The Sundermann et al. [SBK+24] dataset mostly contains feature models of software systems. 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.

Table 7.4 relates the upper and lower bounds computed by Algorithm 4 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 4 is greater than the lower bound obtained 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 4 gives us better bounds than Lemma 1. The 'Both' column indicates how often the bounds returned by Algorithm 4 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 7.3). The r_c ratio quantifies the difference between the bounds obtained by using Lemma 1 and the bounds obtained by using Algorithm 4. r_c uses $min(Y_h, |R_{G_U}|)$ and $max(Y_l, |R_{G_P}|)$ because detecting that the bounds returned by Algorithm 4 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 4 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 4 performs poorly on the Sundermann et al. [SBK+24] dataset and on the feature model subset of the Plazar et al. [PAP+19] dataset. On the other hand, Algorithm 4 performs well on the Bayesian network subset of the Lagniez and Marquis [LM17] dataset, as we observe a success rate of 86%. Moreover, we find that the bounds returned by Algorithm 4 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 4, we compare it against ApproxMC 7 [PMY25]. The results of this comparison are presented in Tables 7.5 and 7.6, which highlight differences in accuracy and runtime performance.

Table 7.5 presents the accuracy results. Column #F reports the number of formulae for which the statistics were computed from the results produced by

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 [LM17]	1509	0.868	0.913	0.801	7.53e-9	1.0
↪↪ Bayesian Network	919	0.958	0.905	0.867	1.62e-6	0.0761
↪↪ BMC	4	1.000	1.000	1.000	0.0	6.67e-5
↪↪ Circuit	58	0.914	0.862	0.828	1.87e-29	8.88e-8
↪↪ Configuration	23	0.522	0.652	0.478	2.58e-6	1.0
↪↪ Handmade	35	1.000	1.000	1.000	0.0	0.0
↪↪ Planning	363	0.887	0.934	0.851	3.63e-42	0.0103
↪↪ QIF	5	0.400	1.000	0.400	2.91e-57	8.9e-6
↪↪ Random	102	0.020	0.961	0.020	1.85e-68	2.01e-23
↪↪ Scheduling	0					
↪ Soos [Soo24]	1191	0.940	0.915	0.863	2.55e-7	0.085
↪ Plazar [PAP+19]	384	0.763	0.815	0.656	6.61e-14	0.0137
↪↪ Blasted Real	173	0.942	0.971	0.913	1.06e-13	0.0137
↪↪ Feature Models	115	0.348	0.478	0.087	1.99e-13	0.00868
↪↪ V15	23	1.000	1.000	1.000	1.37e-16	0.00121
↪↪ V3	23	0.957	1.000	0.957	6.68e-17	0.000663
↪↪ V7	23	1.000	0.957	0.957	2.57e-17	0.00365
↪↪ unsorted	27	0.815	0.815	0.630	4.37e-44	8.86e-7
↪ Sundermann [SBK+24]	257	0.249	0.471	0.089	1.6e-13	0.126

Table 7.4: Experimental results comparing bounds obtained with Algorithm 4 and with Lemma 1. Column #F indicates with how many formulae the following statistics have been computed. Column $Y_l \leq |R_{G_P}|$ indicates how often the lower bound returned by Algorithm 4 is correct and larger than the lower bound obtained by using Lemma 1. Similarly, column $Y_h \geq |R_{G_U}|$ indicates how often the upper bound returned by Algorithm 4 is correct and better than the upper bound obtained by using Lemma 1. The 'Both' column indicates how often both predicates are true. The final 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 7.3.

ApproxMC 7 and Algorithm 4. Thus, #F indicates the number of formulae on which we successfully ran ApproxMC 7, Algorithm 4, 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 4) 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 [PMY25], 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

Dataset	#F	$l \leq Y_{\text{ApproxMC}} \leq h$	$l \leq Y \leq h$
Global	2782	0.977	0.881
↪ Lagniez [LM17]	1386	1.000	0.882
↪↪ Bayesian Network	915	1.000	0.854
↪↪ BMC	4	1.000	1.000
↪↪ Circuit	50	1.000	0.740
↪↪ Configuration	6	1.000	0.833
↪↪ Handmade	15	1.000	1.000
↪↪ Planning	289	1.000	0.952
↪↪ QIF	5	1.000	1.000
↪↪ Random	102	1.000	0.990
↪↪ Scheduling	0	-	-
↪ Soos [Soo24]	1180	0.970	0.873
↪ Plazar [PAP+19]	203	0.862	0.921
↪↪ Blasted Real	172	1.000	0.930
↪↪ Feature Models	3	1.000	1.000
↪↪ V15	0	-	-
↪↪ V3	0	-	-
↪↪ V7	0	-	-
↪↪ unsorted	28	0.000	0.857
↪ Sundermann [SBK+24]	13	1.000	0.846

Table 7.5: Experimental results comparing the accuracy of Algorithm 4 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 4) is within the indicated bounds, with $l = \frac{|R_F|}{1.2}$ and $h = 1.2 \times |R_F|$.

parameters to ApproxMC. The value of ε matches the value used by Pote et al. [PMY25], while $\delta = 0.1$ is a standard choice in statistical settings. Additionally, we modified Algorithm 4 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 4. 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 7.6 compares the runtimes of ApproxMC and Algorithm 4. To compare

Dataset	#F	#DivKC	$log_{10}(min)$	*mean*	*median*	$log_{10}(max)$
Global	2888	2265	-3.2	124.5	4.8	5.3
↪ Lagniez [LM17]	1436	1179	-2.9	80.5	5.5	4.4
↪↪ Bayesian Network	964	782	-2.8	32.8	5.0	4.4
↪↪ BMC	4	0	-2.8	-	-	-
↪↪ Circuit	50	34	-2.6	3.5	2.5	1.3
↪↪ Configuration	6	6	1.2	515.1	90.6	3.3
↪↪ Handmade	15	15	0.0	121.2	9.9	3.1
↪↪ Planning	290	237	-2.9	236.4	5.7	4.1
↪↪ QIF	5	4	-0.1	7.3	7.4	1.2
↪↪ Random	102	101	-0.2	100.4	68.2	2.5
↪↪ Scheduling	0	0	-0.2	-	-	-
↪ Soos [Soo24]	1235	953	-3.0	29.6	4.5	4.4
↪ Plazar [PAP+19]	204	122	-3.2	2.7	1.7	1.5
↪↪ Blasted Real	173	113	-2.3	2.8	2.6	1.0
↪↪ Feature Models	3	3	0.4	13.1	7.6	1.5
↪↪ V15	0	0	0.4	-	-	-
↪↪ V3	0	0	0.4	-	-	-
↪↪ V7	0	0	0.4	-	-	-
↪↪ unsorted	28	6	-3.2	0.6	0.5	0.3
↪ Sundermann [SBK+24]	13	11	-0.6	15904.5	29.6	5.3

Table 7.6: Experimental results comparing the runtime of Algorithm 4 with ApproxMC 7 for 20 runs. To limit the computation time, we run ApproxMC 4 times on each formula and multiply the sum of execution times by 5. Similarly, we ran Algorithm 4 4 times (and the compilation process only once) and multiplied the runtime of Algorithm 4 by 5. 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.

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 4, we ran the compilation phase once and then executed the algorithm 4 times per formula; the total runtime of Algorithm 4 was also multiplied by 5 to simulate 20 runs. The reported execution time for Algorithm 4 includes one call to DivKC and 20 simulated runs of Algorithm 4. 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 4. 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 4 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 in Chapter 6 to test our sampler presented in Algorithm 5. Algorithm 5 is run with $k = 50$ and with batch sizes of $N = 1000$, similarly to the experimental setup in Chapter 6. We used the same dataset (which we called the Ω dataset) to facilitate the comparison with our results. We proposed five tests, of which we used only four, as the last test faces scalability issues according to our original experiments. The simplest test is the modbit test. With our sampler, we obtained a Harmonic mean p-value of 0.13 for the modbit 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 Chapter 6 passed a single test on the dataset. We also used the VF, SFpC, and Birthday tests, which were successfully performed on 176, 77, and 140 formulae, respectively. 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 5 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 in Chapter 6 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 Chapter 6. By doing so, we obtain results that are comparable with the results in Chapter 6. Table 7.7 shows a reproduction of the results in Chapter 6. We extended the table with our results for Algorithm 5.

	Uniformity		Observed number of repetitions			
Sampler	#F	p-value	min	max	average	median
KUS	142	0.001	0	18	9.01	10
QuickSampler	139	0.000	0	29858	480.01	4
Smarch	78	**0.023**	3	22	10.71	10
SPUR	145	0.000	4	307	34.48	12
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
BDDSampler	92	**0.274**	3	17	9.96	10
K-Sampler	140	0.005	3	26	10.75	10

Table 7.7: Extended experimental results for the birthday test with the Ω dataset introduced in Chapter 6 and extended with our results for Algorithm 5. 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).

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 [DLB+18] and CMSGen [GSC+21]. 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 we tested in Chapter 6.

7.4 Conclusion

In this chapter, 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 [LM17] 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 Chapter 6. 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 [PAP+19]. 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.

8

Conclusion

This final chapter summarises the main findings of the dissertation and proposes avenues for future research.

Contents

The main objective of this thesis was to study and enhance the scalability of uniform random sampling. In addition, we also provide a set of five statistical tests designed to evaluate the quality of uniform random samplers and to accelerate their development.

8.1 Summary of Contributions

Empirical studies on the practical complexity of URS. To effectively improve the scalability of URS in practice, we first needed to understand why URS performs well on some formulae but poorly on other seemingly similar formulae. To this end, we conducted two empirical studies exploring the impact of structural metrics on the practical complexity of URS.

Our findings reveal that metrics such as the MIS size and the number of equivalence classes are highly correlated with the time and memory usage of uniform random samplers.

To reduce the noise caused by the heterogeneity in our dataset, we also investigated phase transitions in synthetic k-CNF formulae. This study uncovered that both URS and #SAT exhibit a phase transition phenomenon similar to that observed in SAT solvers. However, unlike SAT solvers, the phase transition for URS appears to be linked to the solution density $\left(\frac{\log_2(|R_F|)}{|Var(F)|}\right)$.

Furthermore, we found that the empirical complexity of URS is strongly influenced by the number of variables and the community structure of the formula. This insight helps explain why current URS techniques are effective on large industrial formulae, which often exhibit strong community structure.

Statistical tests that can be used to develop and debug uniform random samplers. In addition to our empirical studies, we proposed a set of five statistical tests specifically designed to evaluate and debug uniform random samplers. Applying these tests revealed that several state-of-the-art samplers produce low-quality samples, likely due to subtle implementation bugs.

We further showed how the birthday test can be used to gain insight into heuristic-based samplers, even when they fail traditional statistical tests. By analysing the number of repeated models, we can determine whether the sampler is adequately exploring the model space or if its heuristics require adjustment.

Finally, we demonstrated, using synthetic formulae, that the choice of input formula can introduce bias into evaluation results. Therefore, it is essential to validate uniform random samplers on a diverse set of formulae from different sources to ensure the reliability and generalisability of conclusions.

DivKC: a divide-and-conquer approach to KC which can be used as a basis for a state-of-the-art heuristic-based uniform random sampler. Building on the insights from our empirical studies, we developed DivKC, a divide-and-conquer approach to knowledge compilation (KC). The core idea behind DivKC

is to decompose a complex input formula into smaller sub-formulae, which can then be compiled individually. This modular approach improves both the efficiency and tractability of the compilation process.

Using DivKC, we successfully compiled 114 benchmark formulae to d-DNNF that were previously out of reach for the state-of-the-art compiler D4, highlighting its practical advantage in handling large or structurally challenging instances. This result demonstrates that structural decomposition is not only feasible but also an effective strategy for overcoming some of the limitations of state-of-the-art compilers.

On top of DivKC, we implemented an approximate model counter and a heuristic-based uniform random sampler. We evaluated the model counter on a diverse set of benchmarks and observed strong performance.

Finally, we assessed the uniform random sampler using the statistical test suite introduced in Chapter 6. The results show that our sampler consistently outperforms existing heuristic-based approaches, passing a greater number of statistical tests and producing higher-quality samples. These findings suggest that DivKC provides a robust foundation for building scalable and statistically reliable uniform random samplers.

8.2 Perspectives

As demonstrated by DivKC, partial compilation is a promising approach to overcoming some of the scalability limitations of exact knowledge compilation. We have shown that DivKC not only improves scalability but also provides a solid foundation for developing more effective heuristic-based samplers.

Building on these results, a natural next step is to further explore partial compilation techniques. One particularly promising direction is to investigate approaches that avoid relying on projected formulae, as is currently done in DivKC. Our experiments suggest that the projection step often represents a significant bottleneck. Eliminating this dependency, for example, by working directly with upper bounds on the model space, could simplify the architecture of the sampler and make the approach applicable to a wider range of formulae.

As part of future work, we plan to study the factors that influence the scalability and quality of upper bounds, and to investigate strategies for constructing them in ways that balance computational efficiency with memory usage.

List of Publications and Tools

Papers Included in the Dissertation

- 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)*, pages 23–32, 2024
- Olivier Zeyen et al. Exploring the Computational Complexity of Uniform Random Sampling and SAT Counting with Phase Transitions. In *Proceedings of the 29th ACM International Systems and Software Product Line Conference*, 2025

Tools Included in the Dissertation

- URS testing framework:
  Olivier Zeyen. Testing Uniform Random Samplers: Methods, Datasets and Protocols. `https://github.com/serval-uni-lu/urs_test`, 2025
- DivKC:
  Olivier Zeyen. DivKC: A Divide-and-Conquer Approach to Knowledge Compilation. `https://github.com/serval-uni-lu/divkc`, 2025

List of Figures

List of Tables

AI Assistance Disclosure

I hereby acknowledge that parts of the text in this thesis were refined with the assistance of ChatGPT (OpenAI, https://chat.openai.com), an AI language model. The tool was used solely for language polishing and rewriting purposes; all ideas, analysis, and conclusions are my own.

Bibliography

[AGL12] 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 (cited on pages 10, 34, 39, 41, 49, 84).

[AHT18] D. Achlioptas, Zayd Hammoudeh, and P. Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *SAT*, 2018 (cited on pages 4, 12, 13, 15, 18, 37, 56).

[AMM22] Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. On the structure of the boolean satisfiability problem: a survey. *ACM Comput. Surv.*, 55(3), March 2022. ISSN: 0360-0300. DOI: 10.1145/3491210. URL: https://doi.org/10.1145/3491210 (cited on pages 14, 18, 30).

[APC21] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. BURST: a benchmarking platform for uniform random sampling techniques. In Mohammad Reza Mousavi and Pierre-Yves Schobbens, editors, *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B*, pages 36–40. ACM, 2021. DOI: 10.1145/3461002.3473070. URL: https://doi.org/10.1145/3461002.3473070 (cited on pages 31, 53, 79).

[BCM+21] Teodora Baluta, Zheng Leong Chua, Kuldeep S. Meel, and Prateek Saxena. Scalable quantitative verification for deep neural networks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 312–323. IEEE, 2021. DOI: 10.1109/ICSE43902.2021.00039. URL: https://doi.org/10.1109/ICSE43902.2021.00039 (cited on page 2).

[BDG+08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity

clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20:172–188, 2008 (cited on page 10).

[BEB18] Robert G Brown, Dirk Eddelbuettel, and David Bauer. Dieharder. *Duke University Physics Department Durham, NC*:27708–0305, 2018 (cited on pages 15, 56).

[BH19] Joseph K. Blitzstein and Jessica Hwang. *Introduction to Probability, Second Edition*. Chapman and Hall/CRC, Boca Raton, 2nd edition edition, February 2019. ISBN: 978-1-138-36991-7 (cited on page 64).

[BL99] Elazar Birnbaum and Eliezer L Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999 (cited on pages 14, 36, 39).

[BMC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In *Proceedings of CAiSE'05*, pages 491–503, 2005 (cited on page 35).

[BP00] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. Counting models using connected components. In *AAAI/IAAI*, pages 157–162, 2000 (cited on pages 14, 39).

[BV21] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.*, 47(4), September 2021. ISSN: 0098-3500. DOI: 10.1145/3460772. URL: https://doi.org/10.1145/3460772 (cited on page 15).

[ÇA11] Ümit V Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). 2011 (cited on page 84).

[CFM+15] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform SAT witness generation. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, pages 304–319, 2015 (cited on pages 30, 53, 79).

[CFM13] Michael Codish, Yoav Fekete, and Amit Metodi. Backbones for equality. In *Haifa Verification Conference*, 2013 (cited on pages 18, 20, 21).

[CLS14] Shaowei Cai, Chuan Luo, and Kaile Su. Scoring functions based on second level score for k-SAT with long clauses. *J. Artif. Intell. Res.*, 51:413–441, 2014. URL: https://api.semanticscholar.org/CorpusID:16385185 (cited on page 34).

[CM19] Sourav Chakraborty and Kuldeep S. Meel. On testing of uniform samplers. In *AAAI Conference on Artificial Intelligence*, 2019. URL: https://api.semanticscholar.org/CorpusID:106401023 (cited on pages 4, 13).

[CMV14] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in sat witness generator. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, 60:1–60:6, San Francisco, CA, USA. ACM, 2014. ISBN: 978-1-4503-2730-5. DOI: 10.1145/2593069.2593097. URL: http://doi.acm.org/10.1145/2593069.2593097 (cited on pages 8, 14, 30, 53, 79).

[Dar+04] Adnan Darwiche et al. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332. Citeseer, 2004 (cited on page 12).

[Dar00] Adnan Darwiche. On the tractable counting of theory models and its application to belief revision and truth maintenance. *arXiv preprint cs/0003044*, 2000 (cited on page 82).

[DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008 (cited on pages 18, 23).

[DD06] Gilles Dequen and Olivier Dubois. An efficient approach to solving random k-SAT problems. *Journal of Automated Reasoning*, 37:261–276, 2006. URL: https://api.semanticscholar.org/CorpusID:25414864 (cited on page 34).

[DLB+18] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 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*, pages 549–559, 2018. DOI: 10.1145/3180155.3180248. URL: http://doi.acm.org/10.1145/3180155.3180248 (cited on pages 13, 14, 24, 30, 37, 53, 66, 79, 90, 99).

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002 (cited on pages 82, 84).

[Dot] Chris Doty-Humphrey. Practically random. https://sourceforge.net/projects/pracrand/. Accessed: 2024-10-14 (cited on pages 15, 56).

[dPDD15] Axel de Perthuis de Laillevault, Benjamin Doerr, and Carola Doerr. Money for nothing: speeding up evolutionary algorithms through better initialization. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 815–822, Madrid, Spain. ACM, 2015. ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754760. URL: http://doi.acm.org/10.1145/2739480.2754760 (cited on page 2).

[EGS12] 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, pages 255–264, Catalina Island, CA. AUAI Press, 2012. ISBN: 9780974903989 (cited on pages 3, 13, 15, 56).

[EO22] Guillaume Escamocher and Barry O'Sullivan. Generation and prediction of difficult model counting instances. *ArXiv*, abs/2212.02893, 2022 (cited on pages 14, 67).

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT'03*, pages 502–518. Springer, 2003 (cited on pages 18, 23, 43).

[FHH20] Johannes Klaus Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *Journal of Experimental Algorithmics (JEA)*, 26:1–26, 2020 (cited on pages 12, 37).

[GBC+02] Ivo Grosse, Pedro Bernaola-Galván, Pedro Carpena, Ramón Román-Roldán, Jose Oliver, and H Eugene Stanley. Analysis of symbolic sequences using the jensen-shannon divergence. *Physical Review E*, 65(4):041905, 2002 (cited on page 63).

[GHS+07] Carla P Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI*, volume 2007, pages 2293–2299, 2007 (cited on page 12).

[GL15] Jesús Giráldez-Cru and Jordi Levy. A modularity-based random sat instances generator. In *International Joint Conference on Artificial Intelligence*, 2015 (cited on pages 9, 10, 36, 39–41).

[GLY17] Jian Gao, Ruizhi Li, and Minghao Yin. A randomized diversification strategy for solving satisfiability problem with long clauses. *Science China Information Sciences*, 60:1–11, 2017. URL: https://api.semanticscholar.org/CorpusID:13219850 (cited on page 34).

[Goo58] I. J. Good. Significance tests in parallel and in series. *Journal of the American Statistical Association*, 53:799–813, 1958. URL: https://api.semanticscholar.org/CorpusID:121267100 (cited on page 59).

[GRM20] Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Phase transition behavior in knowledge compilation. In *International Conference on Principles and Practice of Constraint Programming*, 2020. URL: https://api.semanticscholar.org/CorpusID:220665810 (cited on pages 14, 15, 34–37, 39, 43, 44, 46).

[GSC+21] Priyanka Golia, M. Soos, Sourav Chakraborty, and Kuldeep S. Meel. Designing samplers is easy: the boon of testers. *2021 Formal Methods in Computer Aided Design (FMCAD)*:222–230, 2021 (cited on pages 3, 4, 13, 16, 18, 99).

[GV21] Vijay Ganesh and Moshe Y. Vardi. On the unreasonable effectiveness of sat solvers. In Tim Roughgarden, editor, *Beyond the worst-case analysis of algorithms*, pages 547–563. Cambridge University Press, 2021 (cited on pages 14, 30, 53).

[GW94] Ian P. Gent and Toby Walsh. The SAT phase transition. In *European Conference on Artificial Intelligence*, volume 94, pages 105–109. PITMAN, 1994 (cited on page 34).

[GYX11] Jian Gao, Minghao Yin, and Ke Xu. Phase transitions in knowledge compilation: an experimental study. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 364–366. Springer, 2011 (cited on pages 14, 34, 36, 39).

[HD04] Jinbo Huang and Adnan Darwiche. Using dpll for efficient obdd construction. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–172. Springer, 2004 (cited on pages 14, 36, 39).

[HFG+20] Ruben Heradio, David Fernández-Amorós, José A. Galindo, and David Benavides. Uniform and scalable sat-sampling for configurable systems. In Roberto Erick Lopez-Herrejon, editor, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, 17:1–17:11. ACM, 2020. DOI: 10.1145/3382025.3414951. URL: https://doi.org/10.1145/3382025.3414951 (cited on pages 62, 63).

[HFG+22] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, David Benavides, and Don Batory. Uniform and scalable sampling of highly configurable systems. *Empirical Softw. Engg.*, 27(2), March 2022. ISSN: 1382-3256. DOI: 10.1007/s10664-021-10102-5. URL: https://doi.org/10.1007/s10664-021-10102-5 (cited on pages 3, 4, 13, 16, 69).

[HFM+19] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. Supporting the statistical analysis of variability models. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 843–853. IEEE, 2019 (cited on page 68).

[HNA+18] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. *Empirical Software Engineering*, July 2018. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9635-4. URL: https://doi.org/10.1007/s10664-018-9635-4 (cited on pages 2, 24, 31, 53, 66, 79).

[HS15] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23:1–34, 2015. URL: https://api.semanticscholar.org/CorpusID:3395784 (cited on page 24).

[IMM+16] Alexander Ivrii, Sharad Malik, Kuldeep S Meel, and Moshe Y Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016 (cited on pages 8, 20, 25).

[JHF11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS '11*, number Section 3, pages 638–652. Springer, 2011. ISBN: 978-3-642-24485-8 (cited on page 14).

[Ken38] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1-2):81–93, 1938 (cited on page 41).

[KPN09] Aaron A. Klammer, Christopher Y. Park, and William Stafford Noble. Statistical calibration of the sequest xcorr function. *Journal of Proteome Research*, 8(4):2106–2113, 2009. DOI: 10.1021/pr8011107. eprint: https://doi.org/10.1021/pr8011107. URL: https://doi.org/10.1021/pr8011107. PMID: 19275164 (cited on page 78).

[KTM+17] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 291–302, 2017. DOI: 10.1145/3106237.3106252. URL: http://doi.acm.org/10.1145/3106237.3106252 (cited on pages 24, 66).

[KTS+18] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating configuration decisions with modal implication graphs. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 898–909, 2018. DOI: 10.1145/3180155.3180159. URL: http://doi.acm.org/10.1145/3180155.3180159 (cited on pages 24, 66).

[LGC+15] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, pages 91–100, Nashville, Tennessee. ACM, 2015. ISBN: 978-1-4503-3613-0. DOI: 10.1145/2791060.2791070. URL: http://doi.acm.org/10.1145/2791060.2791070 (cited on pages 14, 24, 49, 66).

[Lin91] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991. DOI: 10.1109/18.61115 (cited on page 63).

[LL24] 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 (cited on page 86).

[LLM16] 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 (cited on pages 8, 83).

[LM17] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *IJCAI*, 2017 (cited on pages 4, 12, 18, 20, 23, 37, 45, 49, 53, 82, 88, 90–97, 99).

[LMS18] Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. Dmc: a distributed model counter. In *27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, pages 1331–1338, 2018 (cited on page 12).

[LS07] Pierre L'Ecuyer and Richard Simard. Testu01: a c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):1–40, 2007 (cited on pages 15, 56, 69).

[Mat11] R. Mateescu. Treewidth in industrial sat benchmarks. In 2011 (cited on page 14).

[MH15] Zongxu Mu and Holger H. Hoos. On the empirical time complexity of random 3-SAT at the phase transition. In *International Joint Conference on Artificial Intelligence*, 2015. URL: https://api.semanticscholar.org/CorpusID:17251755 (cited on page 34).

[MMB+12] Christian Muise, Sheila A McIlraith, J Christopher Beck, and Eric I Hsu. 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*, pages 356–361. Springer, 2012 (cited on page 12).

[MPC20] Kuldeep S. Meel, Yash Pote, and Sourav Chakraborty. On testing of samplers. In *Advances in Neural Information Processing Systems(NeurIPS)*, December 2020 (cited on page 68).

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of sat problems. In *AAAI Conference on Artificial Intelligence*, 1992 (cited on pages 34–36, 38, 39, 43, 44, 67).

[MWC09] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, San Francisco, California, USA. Carnegie Mellon University, 2009. URL: http://dl.acm.org/citation.cfm?id=1753235.1753267 (cited on pages 14, 35, 50).

[MZK+99] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, 1999 (cited on page 14).

[NG03] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69 2 Pt 2:026113, 2003 (cited on page 10).

[OBM+17] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 61–71. ACM, 2017. DOI: 10.1145/3106237.3106273. URL: https://doi.org/10.1145/3106237.3106273 (cited on page 2).

[OD14] Umut Oztok and Adnan Darwiche. On compiling cnf into decision-dnnf. In *CP*, 2014 (cited on page 20).

[OGB+20] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. Scalable Uniform Sampling for Real-World Software Product Lines. Technical report TR-20-01, 2020 (cited on pages 3, 12, 16).

[OGB19] Jeho Oh, Paul Gazzillo, and Don S. Batory. t-wise coverage by uniform sampling. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, 15:1–15:4. ACM, 2019. ISBN: 978-1-4503-7138-4 (cited on page 2).

[ONe18] Melissa E. O'Neill. A birthday test: quickly failing some popular prngs. `https://www.pcg-random.org/posts/birthday-test.html`, 2018. Accessed: 2024-10-14 (cited on page 64).

[PAP+19] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 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*, pages 240–251, 2019 (cited on pages 2, 3, 13–15, 18, 24, 31, 37, 53, 60, 61, 66, 79, 90–97, 100).

[Pea00] Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 1*, 50:11–28, 1900. URL: `https://api.semanticscholar.org/CorpusID:123466489` (cited on page 60).

[PJM19] Yash Pote, Saurabh Joshi, and Kuldeep S. Meel. Phase transition behavior of cardinality and XOR constraints. *ArXiv*, abs/1910.09755, 2019. URL: `https://api.semanticscholar.org/CorpusID:199465708` (cited on page 34).

[PMY25] Yash Pote, Kuldeep S Meel, and Jiong Yang. Towards real-time approximate counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39 of number 11, pages 11318–11326, 2025 (cited on pages 12, 82, 94–96).

[PS19] Daniël Paulusma and Stefan Szeider. On the parameterized complexity of (k, s)-sat. *Inf. Process. Lett.*, 143:34–36, 2019 (cited on page 20).

[PTR+19] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. Product sampling for product lines: the scalability challenge. *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, 2019. URL: `https://api.semanticscholar.org/CorpusID:85462202` (cited on page 20).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011 (cited on page 24).

[Rai15] Matt Raible. *The JHipster mini-book*. C4Media, 2015 (cited on pages 24, 66).

[RSN+01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, volume 22. US Department of Commerce, Technology Administration, National Institute of ..., 2001 (cited on pages 15, 56, 78).

[SBK+24] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference*, pages 54–65, New York, NY, USA. ACM, September 2024 (cited on pages 37, 53, 90–97).

[SGC+22] Mate Soos, Priyanka Golia, Sourav Chakraborty, and Kuldeep S. Meel. On quantitative testing of uniform samplers. In *Proceedings of International Conference on Constraint Programming (CP)*, August 2022 (cited on pages 15, 57).

[SGM20] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*, July 2020 (cited on pages 3, 13, 15, 18, 37, 56).

[SGR+18] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Knowledge compilation meets uniform sampling. In *Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, November 2018 (cited on pages 3, 4, 13, 15, 23, 82).

[SHN+23] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. Evaluating state-of-the-art # sat solvers on industrial configuration spaces. *Empirical Software Engineering*, 28, 2023 (cited on pages 12, 14, 37, 82).

[SM21] Mate Soos and Kuldeep S Meel. Arjun: an efficient independent support computation technique and its applications to counting and sampling. *arXiv preprint arXiv:2110.09026*, 2021 (cited on pages 20, 23, 25, 26, 31, 83).

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. DOI: 10.1007/978-3-642-02777-2_24. URL: https://doi.org/10.1007/978-3-642-02777-2%5C_24 (cited on page 13).

[Soo24] Mate Soos. Benchmarks used for approximate model counting. Zenodo, January 2024. DOI: 10.5281/zenodo.10449477. URL: https://doi.org/10.5281/zenodo.10449477 (cited on pages 37, 90–93, 95–97).

[SRH+24] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. Reusing d-dnnfs for efficient feature-model counting. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–32, 2024 (cited on pages 12, 82).

[SRS+19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. Ganak: a scalable probabilistic exact model counter. In *IJCAI*, volume 19 of number 2019, pages 1169–1176, 2019 (cited on page 12).

[Thu06] Marc Thurley. Sharpsat – counting models with advanced component caching and implicit bcp. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 424–429, Berlin, Heidelberg. Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-37207-3 (cited on pages 4, 12, 13, 20, 23, 37).

[Val79] Leslie G Valiant. The complexity of enumeration and reliability problems. *siam Journal on Computing*, 8(3):410–421, 1979 (cited on pages 4, 12, 15).

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985 (cited on page 13).

[Wan15] Yisong Wang. On forgetting in tractable propositional fragments. *ArXiv*, abs/1502.02799, 2015. URL: https://api.semanticscholar.org/CorpusID:6588613 (cited on page 8).

[Wil19] Daniel J. Wilson. The harmonic mean p-value for combining dependent tests. *Proceedings of the National Academy of Sciences of the United States of America*, 116:1195–1200, 2019. URL: https://api.semanticscholar.org/CorpusID:58589423 (cited on page 59).

[WS05] 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*, pages 324–339. Springer, 2005 (cited on page 12).

[Yat34] F. Yates. Contingency tables involving small numbers and the $\chi 2$ test. In 1934. URL: https://api.semanticscholar.org/CorpusID:126200663 (cited on pages 60, 61, 68).

[YM23] Jiong Yang and Kuldeep S Meel. Rounding meets approximate model counting. In *International Conference on Computer Aided Verification*, pages 132–162. Springer, 2023 (cited on page 12).

[ZCP+24] Olivier Zeyen, Maxime Cordy, Gilles Perrouin, and Mathieu Acher. 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)*, pages 23–32, 2024 (cited on page iii).

[ZCP+25] Olivier Zeyen, Maxime Cordy, Gilles Perrouin, and Mathieu Acher. Exploring the Computational Complexity of Uniform Random Sampling and SAT Counting with Phase Transitions. In *Proceedings of the 29th ACM International Systems and Software Product Line Conference*, 2025 (cited on page iii).

[Zey23] Olivier Zeyen. Preprocessing is What You Need: Understanding and Predicting the Complexity of SAT-based Uniform Random Sampling. https://github.com/serval-uni-lu/urs_scal, 2023 (cited on page 25).

[Zey25a] Olivier Zeyen. DivKC: A Divide-and-Conquer Approach to Knowledge Compilation. https://github.com/serval-uni-lu/divkc, 2025 (cited on pages 83, iii).

[Zey25b] Olivier Zeyen. Exploring the Computational Complexity of SAT Counting and Uniform Sampling with Phase Transitions. `https://github.com/serval-uni-lu/urs_phase_transitions`, 2025 (cited on pages 38, 39, 50, 53).

[Zey25c] Olivier Zeyen. Testing Uniform Random Samplers: Methods, Datasets and Protocols. `https://github.com/serval-uni-lu/urs_test`, 2025 (cited on pages 67, 69, 80, iii).