When to Sample from Feature Diagrams?

Nikolai Käfer
Dresden University of Technology
Dresden, Germany
nikolai.kaefer@tu-dresden.de

Sven Apel
Saarland University
Saarbrücken, Germany
apel@cs.uni-saarland.de

Christel Baier
Dresden University of Technology
Dresden, Germany
christel.baier@tu-dresden.de

Clemens Dubslaff
Eindhoven University of Technology
Eindhoven, The Netherlands
c.dubslaff@tue.nl

Holger Hermanns
Saarland University
Saarbrücken, Germany
hermanns@cs.uni-saarland.de

Abstract

Uniform random sampling (URS) has many applications in configurable systems analysis. Usually, feature models consisting of a hierarchical feature diagram and additional side constraints specify the space of valid configurations to be sampled from. However, URS has predominately been applied on feature models translated *a priori* into conjunctive normal form (CNF). In this work, we study URS approaches that instead operate directly on feature diagrams and provide a comparative evaluation of their performance against well-established URS tools for CNF representations. Our findings suggest that translating feature models to CNF offers advantages, even in the presence of only few side constraints.

Keywords

Uniform Random Sampling, Feature Models, BDDs

ACM Reference Format:
Nikolai Käfer, Sven Apel, Christel Baier, Clemens Dubslaff, and Holger Hermanns. 2024. When to Sample from Feature Diagrams?. In *Proceedings of 19th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'25).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Acknowledgments

This work was partially supported by the NWO through Veni grant VI.Veni.222.431 and by the DFG under the projects EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660).

1 Introduction

Generating a representative set of instances of a complex system domain is a major challenge across computer science. For instance, in the area of data-driven AI, the input space is exponential in the number of features. This renders machine-learning algorithms only feasible when applied to a small subset of the entire input-output relation. The accuracy of ML classifiers then crucially depends on how representative that subset is with respect to the entire input-output relation [8]. Similarly, in software product line engineering, the space of software configurations is exponential, rendering test case generation by means of system configurations a challenging task. Further, the distribution of bugs or other properties of interest is a priori not known and is likely to depend on cross-cutting feature interactions [5, 10]. *Uniform random sampling* (URS) is a key concept to derive representative sample sets with limited information [1, 29] by ensuring that every element of the sample space has the same probability to be sampled. The latter avoids bias in the samples and may allow for establishing at least a certain degree of statistical guarantees.

In this paper we focus on the challenge of URS on configuration spaces of variability-aware software. Configuration spaces are usually given through *feature models* over boolean configuration options, so-called *features* [4, 13]. Most commonly, a feature model comprises a *feature diagram* and additional *side constraints* [4, 23]. A feature diagram is a directed tree over features with certain boolean operators as inner nodes, while side constraints are given as propositional logic formulas or *cross-tree-constraints* annotating the feature diagram. The standard semantics of a feature diagram then induces a propositional logic formula [32]. So, the main benefit of a feature diagram lies in exposing the hierarchical dependencies of features that commonly are reflected in separation of concerns and structured software development processes. While feature diagrams hence carry much structural domain knowledge, the standard approach to URS over feature models starts off with translating feature diagram and side constraints into a single propositional logic formula in *conjunctive normal form* (CNF), followed by subsequent application of context-agnostic but highly optimized URS approaches for CNFs [1, 19, 29].

URS Approaches and Tools. Due to its broad applications, a variety of tools for URS have been proposed in the literature, covering a range of distinct methods. For instance, *knowledge compilation* approaches translate the input CNF into representations that support efficient sampling [14]. Although these methods involve an initial compilation step that can be time- and resource-intensive, the subsequent sampling tends to be fast. Examples include KUS [33], which leverages d-DNNF (deterministic decomposable negation normal form) compilation [14], and BDDSampler [19], which utilizes *reduced ordered binary decision diagrams* (BDDs) [9] as a target for compilation and URS [25, 42]. Hashing-based methods, such as the UniGen family [11, 12, 35], combine random hash functions with satisfiability (SAT) solvers to produce near-uniform samples. Model counting (#SAT) asks for the exact count of satisfying assignments of a CNF, a computational problem that is closely related to URS [22]. For example, SPUR [3] adapts the DPLL procedure of many #SAT

VaMoS'25, February 2-6, 2025, Rennes, France
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

solvers [7] to produce uniform samples during model counting, while Smarch [30] constructs a bijection between solutions and integers within the model count range to achieve perfect uniform sampling. Heradio et al. [19] recently conducted a comparative evaluation of various URS tools applied on feature models given in CNF, but without taking into account the resource-intensive knowledge compilation steps towards generating BDDs or d-DNNFs. As can be expected, this perspective makes knowledge compilation approaches clearly outperform #SAT-based approaches, but leaves open the research question whether and for which sample sizes the resource investments for knowledge compilation pay off:

(RQ1) Which URS approaches and tools perform best on feature models for various sample sizes, also taking knowledge compilation resources into account?

URS on Feature Diagrams. As illustrated above, commonly used URS tools are not specifically tailored to the feature model context. Therefore, to sample from feature models, one must extract constraints from the feature diagram, incorporate any side constraints, and convert the resulting propositional formula to CNF. However, this hinders the exploitation of structural domain knowledge on the configuration space given in the feature diagram. Notably, both model counting [40] and URS [39] can be performed in linear time on feature diagrams (i.e., feature models without side constraints). In the following, we denote by *URS-FD* this URS approach on feature diagrams [39]. While this suggests that *URS-FD* clearly outperforms classical URS via CNFs, which is computational intractable in the worst case [22], it has, to the best of our knowledge, never been implemented and evaluated empirically on real-world feature diagrams. Due to the manifold heuristics and optimizations in state-of-the-art URS tools, this directly raises the question whether the linear-time URS-FD approach is actually as performant as it could be expected from its superior worst-case time complexity.

(RQ2) How competitive is URS-FD compared to URS tools on feature models without side constraints?

Rejection Sampling. This standard method [41] generates samples of a random variable over one dimension by uniformly sampling at random over a product sample space and applying a selector criterion to filter the second dimension. Samples that do not satisfy the criterion are discarded, typically followed by resampling. Rejection sampling is especially useful when incorporating the selector criterion directly into the sampling process is difficult.

URS-FD does a priori not support feature models with side constraints. However, a straight-forward extension of the algorithm can be established through rejection sampling, where the feature diagram serves as the first and the side constraints as second sample dimension. After sampling from the feature diagram, we check if the sample satisfies the side constraints of the feature model. If it does, we keep the sample; if not, we discard it and sample again. This approach, which we abbreviate *URS-FD*$^+$, guarantees uniform samples for the entire feature model, since the *sample space*—containing all configurations valid for the diagram—is a superset of the *solution space* which contains all configurations valid for both the diagram and the side constraints.

To the best of our knowledge, rejection sampling has not yet been applied in the area of URS for feature models. This method adds a further dimension to the URS portfolio for feature models and opens the question whether certain characteristics of feature models render specific URS methods to be superior.

(RQ3) What factors determine which sampling methods perform best for a given feature model?

1.1 Contribution

This paper gives answers to all three research questions stated above, providing the following three main contributions:

- We empirically compare state-of-the-art URS tools on feature models with non-trivial feature diagrams.
- We implement the URS-FD algorithm [39] as well as the new URS-FD$^+$ algorithm and compare them empirically.
- We establish phase transition points on the complexity of the feature models where knowledge-compilation and #SAT-based approaches start to outperform URS-FD$^+$.

We implemented URS-FD and URS-FD$^+$ in a tool called Zampler,[1] together with BDD-based knowledge compilation on side constraints to optimize the rejection sampling process within URS-FD$^+$. Our empirical evaluation is performed on commonly used benchmarks that comprise non-trivial feature diagrams [36], excluding feature diagrams where all information resides in side constraints (inducing a trivial diagram consisting of a single AND-node with all features as its children).

Our experiments towards answering RQ1 reveal that, when including the resources for knowledge compilation, the URS tool SPUR is the fastest sampler across all feature models in CNF, whereas thus far KUS and BDDSampler have been considered to be the fastest according to the literature [19]. In response to RQ2, we report that URS-FD indeed outperforms all state-of-the-art URS tools when not taking side constraint of the feature model into account. The efficiency of our tool Zampler—and even its feasibility—is directly related to the size ratio of sampling space vs. solution space, which we reflect by introducing the notion of *hit rate*. This echoes how strongly the side constraints restrict the solution space. Addressing RQ3, we find that for most models with many side constraints, hit rates are too low for rejection sampling to be feasible, and we identify the tipping points where feasibility breaks down.

1.2 Related Work

Benchmarking activities for URS on feature models include BURST [1, 2], a platform integrating many solvers and uniformity checks, or specific comparisons, such as [43], comparing SPUR and UniGen3. Heß et al. investigate URS as a means to achieve high t-wise coverage [20]. They consider QuickSampler, Smarch, and SPUR on a set of 49 feature models, as well as the t-wise samplers Baital and YASA. The introduction of BDDSampler [19] comes with a comparison across existing samplers KUS, Smarch, SPUR, and UniGen2 (and QuickSampler) on a set of 9 feature models. The measurements reported, however, do not include model construction times for BDDSampler and KUS, thus putting the findings partially into question. The remaining results see SPUR orders-of-magnitude faster than Smarch and UniGen2, the latter two timing out for most

[1] The tool and evaluation results are publicly available at https://doi.org/10.5281/zenodo.14536082

models after 1 hour. Earlier work [18, 31] only looked at Smarch, SPUR, and UniGen2 (and QuickSampler). To the best of our knowledge, we are the first to investigate and benchmark the performance of URS with a focus on feature models equipped with non-trivial diagrams and feature models not provided in CNF.

Our paper focuses on URS methods that provide formal guarantees on the uniformity of the distribution of generated samples. Several other approaches instead trade these guarantees for speed, resulting in approximate URS tools such as QuickSampler [16] and CMSGen [17]. URS is tightly connected to #SAT, an observation made in the seminal article by Jerrum et al. [22]. In [37], #SAT solvers were evaluated on industrial-size feature models. URS appears as a motivating application in need of efficient #SAT solvers, pointing particularly at the algorithm used in Smarch [30]. In [28], CNF transformations like the Tseitin transformation and their impact on feature model analysis have been investigated. While this is crucial for the transformation of feature models into CNF we used in our experiments, their work does not investigate URS.

2 Background

2.1 Feature Models

A *feature diagram* (FD) over a finite set of features $\mathcal{V}$ is a tuple $\mathcal{D} = \langle \mathcal{V}, \mathcal{E}, \mathcal{M}, op \rangle$ where

- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is the set of edges defining a directed tree,
- $\mathcal{M} \subseteq \mathcal{V}$ is the subset of features marked as *mandatory*, and
- $op : \mathcal{V} \to \{\text{AND}, \text{OR}, \text{XOR}, \text{LEAF}\}$ assigns an operator to each node.

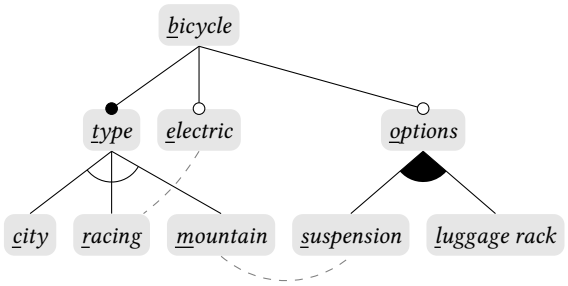
We call a feature diagram $\mathcal{D}$ together with a set C of propositional side constraints over $\mathcal{V}$ a *feature model* (FM).

Let $\overrightarrow{x} := \{y \colon (y, x) \in \mathcal{E}\}$ and $x^{\rightarrow} := \{y \colon (x, y) \in \mathcal{E}\}$ denote, respectively, the set of parents and children of x. A feature diagram $\mathcal{D}$ is *well-formed* if the following conditions hold:

(1) The edge relation $\mathcal{E}$ defines a tree, i.e., there is a designated root node $r \in \mathcal{V}$ s.t. $\overrightarrow{r} = \emptyset$, and for all other nodes $x \in \mathcal{V} \setminus \{r\}$, we have $|\overrightarrow{x}| = 1$.
(2) The operator cardinality is respected for each $x \in Vars$:
   - If $op(x) = \text{LEAF}$, then $|x^{\rightarrow}| = 0$.
   - If $op(x) = \text{AND}$, then $|x^{\rightarrow}| \geq 1$.
   - If $op(x) \in \{\text{OR}, \text{XOR}\}$, then $|x^{\rightarrow}| \geq 2$.
(3) Mandatory nodes may only appear as children of AND nodes, i.e., for all $x \in \mathcal{V}$ where $op(x) \neq \text{AND}$, we have $x^{\rightarrow} \cap \mathcal{M} = \emptyset$.

In the following, we only consider well-formed feature diagrams. The semantics of a feature diagram $\mathcal{D}$ is given as a set of constraints $C_{\mathcal{D}}$ in propositional logic over $\mathcal{V}$ as follows [32]:

- The root r needs to hold: $r \in C_{\mathcal{D}}$.
- Children imply their parents, so for every $x \in \mathcal{V}$ and $c \in x^{\rightarrow}$, we have $(c \to x) \in C_{\mathcal{D}}$.
- AND nodes imply their mandatory children: For $x \in \mathcal{V}$ with $op(x) = \text{AND}$ and every $c \in x^{\rightarrow} \cap \mathcal{M}$, we have $(x \to c) \in C_{\mathcal{D}}$.
- OR nodes imply at least one of their children: For $x \in \mathcal{V}$ with $op(x) = \text{OR}$, we have $\left(x \to \bigvee_{c \in x^{\rightarrow}} c\right) \in C_{\mathcal{D}}$.
- XOR nodes imply exactly one of their children: For $x \in \mathcal{V}$ with $op(x) = \text{XOR}$, we have $\left(x \to \bigvee_{c \in x^{\rightarrow}} c\right) \in C_{\mathcal{D}}$ and $\left(x \to \neg(c \wedge c')\right) \in C_{\mathcal{D}}$ for $c, c' \in x^{\rightarrow}$ s.t. $c \neq c'$.



$$C = \{e \to \neg r, m \to s\}$$

$$C_{\mathcal{D}} = \{b, b \to t, t \to b, e \to b, o \to b,\ t \to (c \vee r \vee m) \wedge \neg(c \wedge r) \wedge \neg(r \wedge m) \wedge \neg(c \wedge m),\ c \to t, r \to t, m \to t, o \to (s \vee l), s \to o, l \to o\}$$

Figure 1: A feature model for a simple bicycle product line (feature diagram $\mathcal{D}$ at the top, set of side constraints C below). The underlined letters are used to abbreviate the feature names. $C_{\mathcal{D}}$ lists the constraints encoded in the diagram.

An example of a feature model is shown in Figure 1, where the root *bicycle* is an AND node, *type* is an XOR node, *options* is an OR node, and all other features are LEAF nodes. Below the AND node *bicycle*, only the child *type* is marked as mandatory. There are two side constraints, highlighted by the dashed lines in the diagram. Intuitively, $e \to \neg r$ states electric bicycles cannot be of type racing, while $m \to s$ assures mountain bikes always come with suspension.

Configurations. A *configuration* $\theta \colon \mathcal{V} \to \mathbb{B}$ is an assignment of each feature variable to either `true` or `false`. The set of all $2^{|\mathcal{V}|}$ possible assignments is denoted by Θ. For a propositional logic formula ϕ, its *valid configurations* are collected in the set $[\![\phi]\!] := \{\theta \colon \theta \models \phi\}$ of all configurations under which ϕ is true. We extend the notation to sets of formulas C in the intuitive way ($[\![C]\!] := \bigcap_{\phi \in C} [\![\phi]\!]$), and—in a slight abuse of notation—to feature diagrams: $[\![\mathcal{D}]\!] := [\![C_{\mathcal{D}}]\!]$. The valid configurations of $\mathcal{F}$ are given by $[\![\mathcal{F}]\!] := [\![\mathcal{D}]\!] \cap [\![C]\!]$. Finally, the *model count* of a structure $\mathcal{S}$ with configuration semantics is defined as the total number of satisfying configurations, i.e., $|[\![\mathcal{S}]\!]|$.

2.2 Model Counting in FDs

The number of valid configurations for a feature diagram $\mathcal{D}$ can be computed recursively with a single pass over $\mathcal{D}$ [40]. For each node $x \in \mathcal{V}$, we recursively define $\#(x)$ as:

$$\#(x) := \begin{cases} 1 & \text{for } op(x) = \text{LEAF} \\ \prod_{c \in x^{\rightarrow} \cap \mathcal{M}} \#(c) \cdot \prod_{c \in x^{\rightarrow} \setminus \mathcal{M}} \left(\#(c) + 1\right) & \text{for } op(x) = \text{AND} \\ \sum_{c \in x^{\rightarrow}} \#(c) & \text{for } op(x) = \text{XOR} \\ \left[\prod_{c \in x^{\rightarrow}} \left(\#(c) + 1\right)\right] - 1 & \text{for } op(x) = \text{OR} \end{cases}$$

Then $\#(x)$ is the model count of the sub-feature diagram rooted in node x. For a feature diagram $\mathcal{D}$ with root node r, we have $\#(r) = |[\![\mathcal{D}]\!]|$.

2.3 Uniform Random Sampling

Let $\Pr[X]$ denote the probability of event X. A *random sampler* $S_{\mathcal{T}}$ for a structure $\mathcal{T}$ is a probabilistic algorithm which produces a random sample $\theta \in [\![\mathcal{T}]\!]$ as output. $S_{\mathcal{T}}$ is *uniform* if for all configurations $\theta \in [\![\mathcal{T}]\!]$, we have

$$\Pr[S_{\mathcal{T}} = \theta] \;=\; \frac{1}{|[\![\mathcal{T}]\!]|}.$$

In [39], a recursive algorithm URS-FD has been provided to dedicatedly sample in feature diagrams uniformly at random. Algorithm 1 shows an adaption of this algorithm, iterating in breadth-first order over the diagram tree to construct a configuration θ. We assume the model count $\#(x)$ has been precomputed for every feature $x \in \mathcal{V}$. $\mathcal{B}_x$ is a boolean random variable with $\Pr[\mathcal{B}_x{=}\texttt{true}] = \frac{\#(x)}{\#(x)+1}$. `weighted_choice`$(\mathcal{W})$ is a probabilistic subroutine that takes a set $\mathcal{W} \subseteq \mathcal{V} \times [0,1]$ of elements and their respective weights and randomly selects one element with probability proportional to its weight. That is, for $(x,w) \in \mathcal{W}$, element x is chosen with probability $\frac{w}{w_{\text{total}}}$ where $w_{\text{total}} = \sum_{(x',w')\in\mathcal{W}} w'$. Note that in line 11, the algorithm loops until at least one child $c \in \vec{x}$ is selected (which happens almost-surely). This is not an issue in practice: since $\Pr[\mathcal{B}_c{=}\texttt{true}] \geq \frac{1}{2}$ for every c, for each loop iteration, the probability of selecting no child and having to repeat the loop is at most $(\frac{1}{2})^n$ for $n = |\vec{x}|$ the number of children.

Algorithm 1: `urs-fd`$(\mathcal{D})$

input : feature diagram $\mathcal{D} = \langle \mathcal{V}, \mathcal{E}, \mathcal{M}, op\rangle$
output : uniform random sample $\theta\colon \mathcal{V} \to \mathbb{B}$

1 $\theta(root) \coloneqq \texttt{true}$
2 **for** $x \in \mathcal{V}$ *in breadth-first order* **do**
3 **if** $\theta(x) = \texttt{false}$ **then**
4 **for** $c \in \vec{x}$ **do** $\theta(c) \coloneqq \texttt{false}$
5 **else**
6 **if** $op(x) = \textsc{and}$ **then**
7 **for** $c \in \vec{x} \cap \mathcal{M}$ **do** $\theta(c) \coloneqq \texttt{true}$
8 **for** $c \in \vec{x} \setminus \mathcal{M}$ **do** $\theta(c) \coloneqq \mathcal{B}_c$
9 **if** $op(x) = \textsc{or}$ **then**
10 $\mathcal{S} \coloneqq \emptyset$
11 **while** $\mathcal{S} = \emptyset$ **do**
12 $\mathcal{S} \coloneqq \left\{c \in \vec{x}\colon \mathcal{B}_c = \texttt{true}\right\}$
13 **for** $c \in \mathcal{S}$ **do** $\theta(c) \coloneqq \texttt{true}$
14 **for** $c \in \vec{x} \setminus \mathcal{S}$ **do** $\theta(c) \coloneqq \texttt{false}$
15 **if** $op(x) = \textsc{xor}$ **then**
16 $c' \coloneqq \texttt{weighted\_choice}\left(\left\{(c, \#(c))\colon c \in \vec{x}\right\}\right)$
17 $\theta(c') \coloneqq \texttt{true}$
18 **for** $c \in \vec{x} \setminus \{c'\}$ **do** $\theta(c) \coloneqq \texttt{false}$
19 **return** θ

3 Rejection Sampling

In this section, we extend the feature diagram sampling algorithm URS-FD to take a feature model's side constraints into account. To this end, we use *rejection sampling*, a generally applicable approach to uniform random sampling, and call the resulting algorithm URS-FD$^+$. Given a uniform random sampler $S_{\mathcal{T}}$ which allows sampling from a *sampling space* $[\![\mathcal{T}]\!]$, we can construct a *rejection sampler* $S_{\mathcal{T} \triangleright \mathcal{T}'}$ for every *solution space* $[\![\mathcal{T}']\!] \subseteq [\![\mathcal{T}]\!]$ as follows: draw candidate samples θ from $S_{\mathcal{T}}$ and return the first satisfying $\theta \in [\![\mathcal{T}']\!]$. Then $S_{\mathcal{T}'}$ is guaranteed to be uniform as well.

The practicality of rejection sampling hinges on two factors: how quickly can we generate and test the candidate samples, and how many candidates are rejected until a valid sample is found. We call the former quantity the *throughput* (*tp*), and define it as the product of the base sampler's sampling rate (*sr*) and the average time t_{check} to test a candidate sample:

$$tp(S_{\mathcal{T} \triangleright \mathcal{T}'}) \coloneqq sr(S_{\mathcal{T}}) \cdot t_{\text{check}}.$$

The second quantity is captured by the *hit rate*, the quotient of the size of solution space and the size of the sample space:

$$hit(S_{\mathcal{T} \triangleright \mathcal{T}'}) \coloneqq \frac{|[\![\mathcal{T}']\!]|}{|[\![\mathcal{T}]\!]|}.$$

Multiplied together, throughput and hit rate define the *effective sampling rate* (*esr*) of a rejection sampler $S_{\mathcal{T} \triangleright \mathcal{T}'}$:

$$esr(S_{\mathcal{T} \triangleright \mathcal{T}'}) \coloneqq hit(S_{\mathcal{T} \triangleright \mathcal{T}'}) \cdot tp(S_{\mathcal{T} \triangleright \mathcal{T}'}). \tag{1}$$

In the context of feature models, our solution space is the set $[\![\mathcal{F}]\!]$ of all valid configurations for the feature model $\mathcal{F}$. An immediate option for a sampling space is the set of all configurations Θ: Clearly, $[\![\mathcal{F}]\!] \subseteq \Theta$ holds, and we get a uniform random sampler S_Θ by constructing assignments with a 50/50 chance of assigning `true` or `false` for every feature. The hit rate for the resulting sampler $S_{\Theta \triangleright \mathcal{F}}$ is $hit(S_{\Theta \triangleright \mathcal{F}}) = |[\![\mathcal{F}]\!]|/2^{|\mathcal{V}|}$. Due to the exponent in the denominator, we expect hit rates to be low for most constrained systems.

The feature diagram sampler $S_{\mathcal{D}}$ from Section 2.3 is also suitable as a base sampler. Since $[\![\mathcal{F}]\!] = [\![\mathcal{D}]\!] \cap [\![\mathcal{C}]\!]$, $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{D}]\!]$ holds, so we can construct the rejection sampler $S_{\mathcal{D} \triangleright \mathcal{F}}$. As the candidate samples already satisfy the diagram constraints by construction, checking the side constraints is sufficient for rejection. The hit rate for diagram-based rejection sampling is $hit(S_{\mathcal{D} \triangleright \mathcal{F}}) = |[\![\mathcal{F}]\!]|/|[\![\mathcal{D}]\!]|$.

Reversing the roles is imaginable as well, namely sampling from the side constraints and rejecting samples that are not valid for the feature diagram. While this still requires a full-fledged constraint sampler, there might be advantages as only a subset of all features usually appears in the side constraints. For example, a BDD-based sampler might be unable to construct a BDD for the whole model within the given memory and time limits, but could manage to construct a BDD for just the side constraints. Yet, the candidate samples still need to cover all features $\mathcal{V}$. Let $\mathcal{V}_C$ denote the subset of features appearing in the side constraints in C, and $[\![C]\!]_{\mathcal{V}_C}$ the set of valid partial samples over the features in $\mathcal{V}_C$. Then all features in $\mathcal{V} \setminus \mathcal{V}_C$ can be freely assigned, which exponentially increases the size of the sample space $[\![C]\!]$. The hit rate of the side constraint-based rejection sampler $S_{C \triangleright \mathcal{F}}$ is thus given by

$$hit(S_{C \triangleright \mathcal{F}}) = \frac{|[\![\mathcal{F}]\!]|}{|[\![C]\!]|} = \frac{|[\![\mathcal{F}]\!]|}{|[\![C]\!]_{\mathcal{V}_C}| \cdot 2^{|\mathcal{V} \setminus \mathcal{V}_C|}}.$$

3.1 Implementation

We implemented two variants of the diagram-based rejection sampler $S_{\mathcal{D}\to\mathcal{F}}$ in Rust. The first variant, Zampler-RS, is straightforward: candidate samples θ are generated using Algorithm 1 and checked against the side constraints by evaluating each constraint $\phi \in C$ directly under the assignment θ.

The second variant, Zampler-RS-BDD, seeks to reduce the time needed for solution checking by constructing a binary decision diagram for the conjunction of all $\phi \in C$. This introduces initial overhead for BDD construction, but allows to check each sample in time linear in the number of features. Additionally, Algorithm 1 is adapted to perform satisfiability checks on partial samples during generation, allowing early rejection before samples are fully constructed. Ideally, if the BDD's variable ordering aligns with the feature diagram's breadth-first order, sample creation and checking proceed smoothly together: as Algorithm 1 traverses the diagram, each assignment of a feature involved in side constraints is mirrored in the BDD by following the high or low edge to the appropriate level. However, the BDD's variable ordering is critical for handling larger models efficiently. Preliminary experiments showed that using orderings consistent with the feature diagram's partial order does not scale well. Therefore, we use heuristics [15] and use a buffered approach for checking partial assignments.

4 Evaluation

4.1 Research Questions

The overarching goal of our experiments is to assess the performance of uniform random samplers on feature models with non-trivial feature diagrams, i.e., feature diagrams where the hierarchical structure contains structural domain knowledge. We hence aim to establish a baseline on existing URS approaches for feature models:

(RQ1) Which URS approaches and tools perform best on feature models for various sample sizes, also taking knowledge compilation resources into account?

Given the linear-time complexity of URS-FD, we ask whether URS-FD can meet or exceed the performance of state-of-the-art URS approaches on feature models without side constraints.

(RQ2) How competitive is URS-FD compared to URS tools on feature models without side constraints?

We anticipate URS-FD to outperform general-purpose tools. However, since real-world feature models without any side constraints are uncommon, answering this question primarily serves to estimate the baseline potential for URS-FD$^+$, extending URS-FD to also support side constraints.

Finally, we are looking for factors indicating that one sampler might be preferable over the others for a specific feature model:

(RQ3) What factors determine which sampling methods perform best for a given feature model?

4.2 Experiment Setup

Our experiment setup is driven by our research questions, selecting state-of-the-art URS tools for feature models in CNF to compare with our implementations of URS-FD and URS-FD$^+$ on community benchmarks comprising non-trivial feature diagrams.

URS Tools. We consider the following samplers from the literature, in their order of appearance: SPUR [3], KUS [33], and BDDSampler [19]. Each tool was introduced with claims of outperforming its predecessors. The URS tools Smarch [30] and UniGen3 [35] have shown great advancements when they have been introduced, but their performance has shown to be not competitive with the tools above (see, e.g., [19]). For a comparison with dedicated URS approaches on feature diagrams, we also consider the rejection samplers implementing URS-FD$^+$ described in Section 3.1, Zampler-RS and Zampler-RS-BDD. Additionally, we consider an alternative BDD-based sampler, Zampler-BDD, which constructs the BDD directly from the feature diagram and the side-constraints instead of a CNF. While BDDSampler uses CUDD [34] for BDD compilation, our implementation Zampler-BDD relies on the recently introduced OxiDD library [21].

Benchmark Set. We collected a set of 46 feature models from the literature [6, 24, 27], ranging in size from 12 to 18,616 features and 1 to 3,545 side constraints.[2] All models are provided in the XML format used by FeatureIDE [38] and consist of a non-trivial feature diagram and a collection of side constraints as propositional formulas. Our implementation Zampler operates directly on these XML files. SPUR and KUS require input as a single CNF formula in DIMACS format, which we generated with FeatureIDE for all models in the benchmark set. For BDDSampler, input is in the form of a VAR file listing all features and an EXP file for the constraints, which we extract from the DIMACS files using dimagic [15]. For RQ1, we used the same 46 models and stripped away their side constraints before re-exporting them as XML and DIMACS.

Operationalization. To evaluate competitiveness of the different URS approach, we consider both runtime and the capacity to generate large sample volumes. As part of RQ3, we investigate the rejection sampling approaches in reference to the hit rate and throughput defined in Section 3. All experiments were run on an Intel Core i9-10900K with 64 GB RAM running Ubuntu 22.04. Throughout, a timeout of 10 minutes (600 s) was enforced.

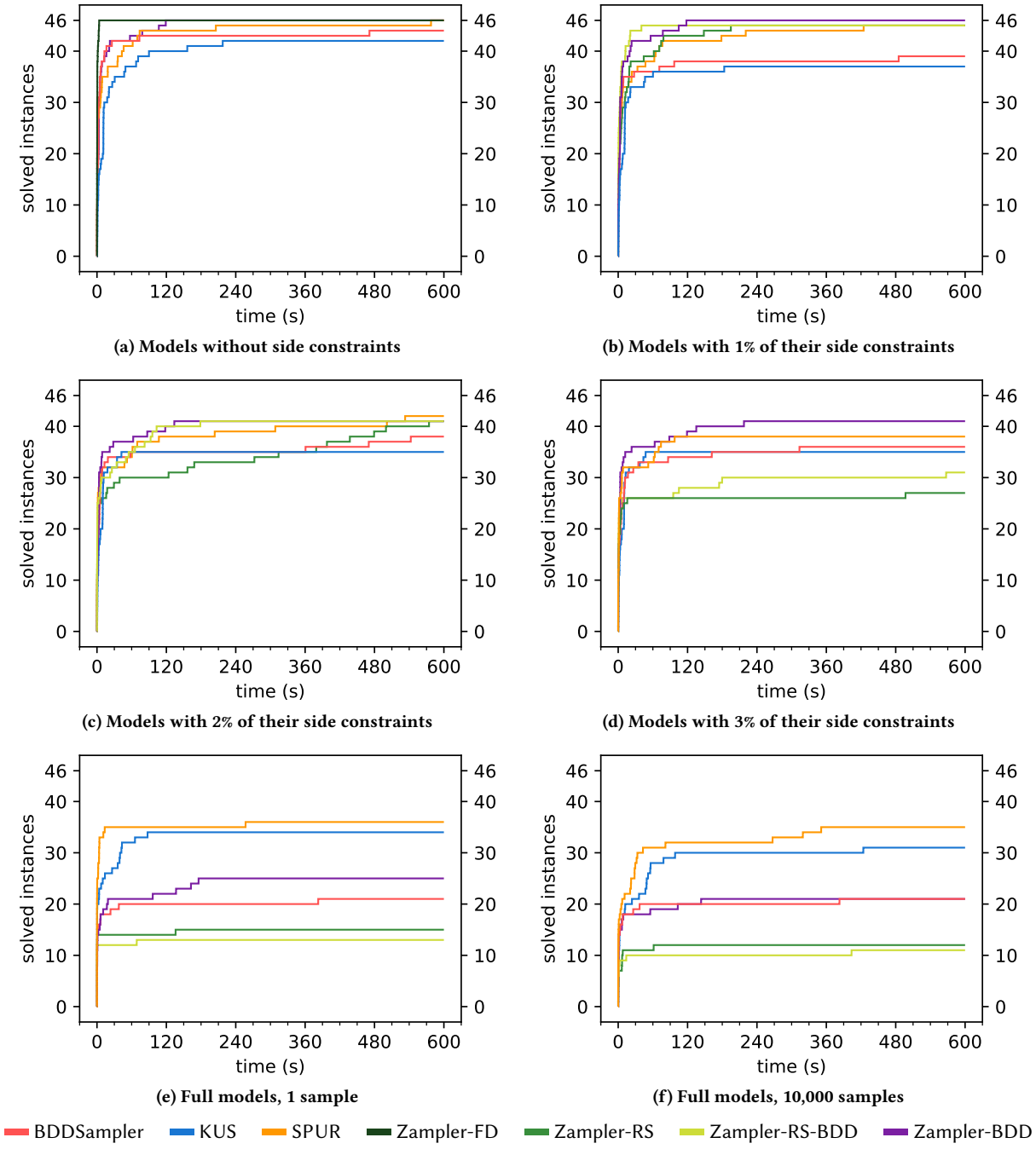
4.3 Results

Figure 2 shows cactus plots for different variants of the benchmark set, indicating for how many of the 46 models the sampling task finished within the time indicated on the x-axis. The task for each solver was to generate 10,000 samples, except for Figure 2e where times to generate a single sample are shown.

4.3.1 Research Question RQ1. Figure 2f shows sampling times for the complete models, i.e., including all side constraints. Clearly, the two rejection sampling methods, Zampler-RS and Zampler-RS-BDD, are not competitive in this setting. Surprisingly, the simpler implementation, Zampler-RS, successfully handles one more model in total than the more optimized Zampler-RS-BDD. Our investigation in the following section, addressing RQ3, will explore possible reasons for this outcome.

Interestingly, KUS was shown in [33] to outperform SPUR across 1,425 benchmarks from various domains, although there is no overlap with our benchmark set. We find the roles reversed, with SPUR

[2] Including 12 artificially generated models from https://github.com/skrieter/MIG-Evaluation/tree/master/de.ovgu.featureide.fm.benchmark/models



(a) Models without side constraints

(b) Models with 1% of their side constraints

(c) Models with 2% of their side constraints

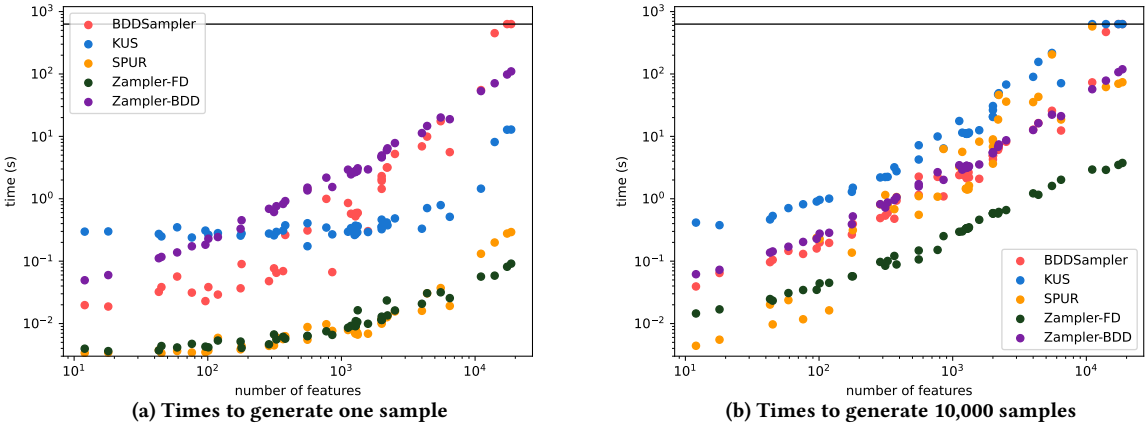
(d) Models with 3% of their side constraints

(e) Full models, 1 sample

(f) Full models, 10,000 samples

Figure 2: Runtime to generate 10,000 samples on models with varying percentage of randomly selected side constraints.

consistently in the lead. The same can be seen in [19] (cf. Table 3), where benchmarks partly overlap with ours. In that table, BDDSampler outperforms SPUR on most models, which contrasts sharply with our findings. This discrepancy is likely due to the use of precompiled BDDs as input for BDDSampler, precluding a fair comparison. The same applies to KUS, where reported timings exclude the d-DNNF compilation step.



(a) Times to generate one sample

(b) Times to generate 10,000 samples

Figure 3: Sampling times in relation to the number of features for models without side constraints.

Figure 2e shows times for generating a single sample. The results show that BDDSampler is largely unaffected by the number of requested samples, suggesting that BDD construction time dominates the total sampling time. For the other samplers, a few additional benchmarks are solvable when only a single sample is requested, although overall performance remains similar regardless of the sample count requested.

Answering **RQ1**, we find SPUR to be the fastest sampler on feature models with side constraints.

4.3.2 Research Question RQ2. Figure 2a presents results for sampling only the feature diagram without side constraints. As expected, Zampler-FD implementing Algorithm 1 outperforms all other solvers, generating 10,000 samples in less than 4 seconds for each model. SPUR succeeds for all benchmarks in 10 minutes, while BDDSampler and KUS time-out for 2 and 4 models, respectively.

We also investigated how model size and the number of requested samples impact performance. The scatter plots in Figure 3 show sampling times for all models, with size on the x-axis and sampling time on the y-axis. The left plot represents single-sample times, while the right plot shows times for generating 10,000 samples. Both axes use a logarithmic scale. We observe that the diagram-based approach Zampler-FD and SPUR are comparably fast when generating a single sample, but with Zampler-FD in the lead for larger sample sizes. Notably, our BDD-based reference implementation Zampler-BDD looks almost indifferent to the number of requested samples. To explore the implementations' limits, we also requested 10^6 samples per model. Zampler-FD is able to generate a million samples for each feature diagram, taking less than 7 minutes for the largest model `Automotive02_V4`. In contrast, KUS and SPUR achieve this only for 18 and 8 of the smaller models, respectively. At this scale, file sizes become substantial enough that I/O performance may influence results, making even larger sample sets less practical to test. Summarizing, we obtain the expected result:

In answer to **RQ2**, we find that Zampler-FD outperforms state-of-the-art URS tools on feature models without side constraints.

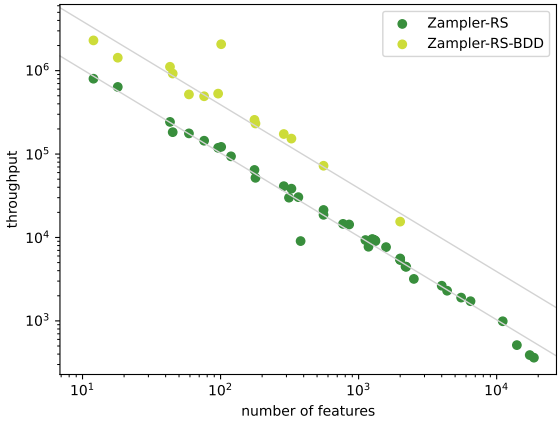


Figure 4: Throughput measures for full models.

4.3.3 Research Question RQ3. Given a feature model, which tool should we use to sample from it? From the discussion of the previous two research questions, we know that SPUR is likely a good overall choice, though Zampler-RS is preferable when our model contains no side constraints and we need many samples. What if we have only a small number of side constraints? Figures 2b to 2d show variants of the benchmark set where only 1%, 2%, or 3% of the models' side constraint were randomly selected and all others removed. While for all solvers the number of solvable benchmarks generally decreases the more side constraints are included, the drop is especially pronounced for the rejection-sampling approaches Zampler-RS and Zampler-RS-BDD.

Taking a step back, we look into the feasibility of the rejection-sampling approaches described in Section 3 in light of the properties of our benchmark models: the feature-based sampler $S_{\Theta \triangleright \mathcal{F}}$, diagram-based sampler $S_{\mathcal{D} \triangleright \mathcal{F}}$, and side constraint-based sampler $S_{\mathcal{C} \triangleright \mathcal{F}}$. Table 1 shows a selection of feature models from the benchmark set along with their number of features $|\mathcal{V}|$, number of side constraints $|C|$, and the percentage of features appearing in the side constraints $|\mathcal{V}_C|/|\mathcal{V}|$. We note that the latter highly varies, ranging

benchmarks	$\lvert\mathcal{V}\rvert$	$\lvert C\rvert$	$\lvert\mathcal{V}_C\rvert$	$\frac{\lvert\mathcal{V}_C\rvert}{\lvert\mathcal{V}\rvert}$	$\log_{10}(\cdot)$				$\log_{10}(hit(\cdot))$			Zampler-RS			Zampler-RS-BDD		
					$\lvert\Theta\rvert$	$\lvert[\![\mathcal{F}]\!]\rvert$	$\lvert[\![\mathcal{D}]\!]\rvert$	$\lvert[\![C]\!]\rvert$	$S_{\mathcal{D}\triangleright\mathcal{F}}$	$S_{C\triangleright\mathcal{F}}$	$S_{\Theta\triangleright\mathcal{F}}$	*tp*	hit^*	out.	*tp*	hit^*	out.
pizzas	12	1	2	16.7%	3.6	1.6	1.7	3.2	-0.1	-1.6	-2.0	800,453.2	-8.7	TP	2,304,322.4	-9.3	TP
GPL	18	13	12	66.7%	5.4	2.6	3.7	4.0	-1.1	-1.3	-2.8	637,867.5	-8.5	TP	1,429,037.2	-9.2	TP
mobilemedia2	43	3	5	11.6%	12.9	6.3	6.7	12.3	-0.3	-6.0	-6.6	242,955.7	-8.2	TP	1,112,852.7	-8.8	TP
jHipster	45	13	25	55.6%	13.5	4.4	5.6	12.1	-1.2	-7.7	-9.1	183,003.7	-8.1	TP	923,299.6	-8.8	TP
aafms_framework	59	14	30	50.8%	17.8	11.1	13.1	15.6	-2.0	-4.4	-6.6	176,967.2	-8.0	TP	519,073.4	-8.7	TP
berkeleyDB1	76	20	32	42.1%	22.9	9.6	14.0	18.1	-4.4	-8.5	-13.3	144,556.3	-7.9	TP	493,043.5	-8.5	TP
KConfig_axTLS	96	14	30	31.3%	28.9	11.9	14.8	26.1	-2.8	-14.2	-17.0	118,824.5	-7.8	TP	530,379.0	-8.4	TP
violet	101	27	67	66.3%	30.4	18.8	26.2	24.5	-7.4	-5.7	-11.6	121,891.0	-7.8	TP	2,074,763.7	-8.4	TP
berkeleyDB2	119	68	97	81.5%	35.8	9.9	29.5	15.8	-19.5	-5.9	-25.9	93,954.3	-7.7	TN		-8.3	TN
BankingSoftware	176	4	5	2.8%	53.0	31.7	31.9	52.4	-0.2	-20.7	-21.3	64,490.4	-7.5	TP	256,989.4	-8.2	TP
WeaFQAs	179	7	13	7.3%	53.9	24.5	25.0	52.8	-0.5	-28.3	-29.4	51,895.4	-7.5	TP	232,485.3	-8.2	TP
eShopSplot	287	21	34	11.8%	86.4	49.4	50.8	83.9	-1.4	-34.5	-37.0	41,223.6	-7.3	TP	173,894.9	-8.0	TP
KConfig_uClibc	313	56	136	43.5%	94.2	40.2	54.3	84.0	-14.1	-43.8	-54.0	30,005.4	-7.3	TN		-7.9	TN
eShopFIDE	326	21	34	10.4%	98.1	49.4	50.8	95.6	-1.4	-46.2	-48.8	38,480.2	-7.3	TP	153,267.2	-7.9	TP
DMIE	366	192	342	93.4%	110.2	97.6	99.0	108.5	-1.3	-10.9	-12.6	30,437.5	-7.2	TP		-7.9	**FP**
KConfig_uClinux-base	380	3,455	299	78.7%	114.4	22.4	22.4	60.0	0.0	-37.6	-92.0	9,056.1	-7.2	TP		-7.8	**FP**
FinancialServices_2017	557	1,001	507	91.0%	167.7	2.6	15.6	134.8	-13.0	-132.2	-165.0	21,366.8	-7.0	TN		-7.7	TN
splot505	557	50	47	8.4%	167.7	65.0	69.2	164.5	-4.2	-99.5	-102.7	18,687.1	-7.0	TP	72,439.7	-7.7	TP
KConfig_busybox-1.18.0	854	123	225	26.3%	257.1	201.3	212.9	243.8	-11.6	-42.5	-55.8	14,316.5	-6.9	TN		-7.5	TN
splot1001	1,120	100	95	8.5%	337.2	125.2	133.1	331.2	-7.9	-205.9	-211.9	9,314.0	-6.7	TN		-7.4	TN
KConfig_embtoolkit	1,179	323	598	50.7%	354.9	96.7	185.8	293.3	-89.1	-196.5	-258.2	7,741.6	-6.7	TN		-7.3	TN
CDL_adder	1,285	890	1,086	84.5%	386.8	124.9	214.1	297.9	-89.2	-173.0	-261.9	9,409.2	-6.7	TN		-7.3	TN
CDL_rattler	1,324	894	1,087	82.1%	398.6	136.5	226.5	309.4	-90.0	-172.9	-262.1	9,066.9	-6.7	TN		-7.3	TN
KConfig_uClinux-distr.	1,580	197	294	18.6%	475.6	409.6	419.5	460.9	-9.8	-51.3	-66.0	7,667.2	-6.6	TN		-7.2	TN
Gen2000-3CNF-1_100	2,000	100	287	14.4%	602.1	329.4	337.0	595.9	-7.6	-266.6	-272.7	5,445.9	-6.5	**FN**	15,494.6	-7.1	**FN**
Automotive_small	2,513	2,833	1,280	50.9%	756.5	210.7	303.4	615.0	-92.7	-404.3	-545.8	3,185.8	-6.4	TN		-7.0	TN
KConfig_linux-2.6.33.3	6,467	3,545	3,994	61.8%	1,946.8	timeout	1,672.6	timeout				1,722.1	-6.0			-6.6	
Automotive02_V1	14,010	666	805	5.7%	4,217.4	1,260.7	1,300.7	4,152.0	-40.0	-2891.3	-2956.7	512.6	-5.6	TN		-6.3	TN
Automotive02_V4	18,616	1,369	1,501	8.1%	5,604.0	1,534.2	1,596.2	5,494.0	-62.0	-3959.8	-4069.7	362.2	-5.5	TN		-6.2	TN

Table 1: Properties, model counts, and hit rate prediction for selected feature models from the benchmark set.

from less than 3% to over 90% of features in the side constraints, and seemingly unrelated to the overall number of features.

To compare the hit rates of the three approaches, we used the model counter SharpSAT-TD [26] to compute the size of the solution space $\lvert[\![\mathcal{F}]\!]\rvert$ and the sample space $\lvert[\![C]\!]\rvert$, and the algorithm shown in Section 2.2 for $\lvert[\![\mathcal{D}]\!]\rvert$. Recall that $\lvert\Theta\rvert = 2^{\lvert\mathcal{V}\rvert}$. The resulting hit rates, computed as defined in Section 3, are also given in Table 1. To allow comparison, all values are given in $\log_{10}$ scale. As expected, $hit(S_{\Theta\triangleright\mathcal{F}})$ is extremely low for all but the smallest models. While much better in comparison, the hit rates for the diagram-based rejection sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$ are still very low for the larger models, providing a plausible reason for the poor performance of the USR-FD$^+$ approaches on the full models. Hit rates of the side-constraint-based rejection sampler $S_{C\triangleright\mathcal{F}}$ fall in between the other two, except for the two models `violet` and `berkeleyDB2`. This indicates $S_{\mathcal{D}\triangleright\mathcal{F}}$ has mostly greater chances to succeed than $S_{C\triangleright\mathcal{F}}$, but averse model characteristics may exist that tip the scale in favor of the latter.

One more outlier sticks out: `KConfig_uClinux-base` has a hit rate of 1 for the diagram sampler $S_{\mathcal{D}\triangleright\mathcal{F}}$, because the sample space $[\![\mathcal{D}]\!]$ and solution space $[\![\mathcal{F}]\!]$ have the same size. As $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{D}]\!]$ holds, this entails that $[\![\mathcal{F}]\!] = [\![\mathcal{D}]\!]$. In other words, the model's 3,455 side constraints only encode constraints that are already implied by the diagram, so $S_{\mathcal{D}\triangleright\mathcal{F}}$ never has to reject a sample.[3]

[3]This explains why `KConfig_uClinux-base` is one of the two models for which Zampler-RS-BDD fails while Zampler-RS succeeds: for the former, the BDD construction times out.

We measured the throughput of Zampler-RS and Zampler-RS-BDD running on the full feature models, i.e., the number of candidate samples generated and tested per second. The respective columns in Table 1 display the average throughput per second after running for 1 minute, with the blank cells for Zampler-RS-BDD indicating the BDD construction timed out after 10 minutes. We find that the early rejection optimization implemented in Zampler-RS-BDD significantly increases throughput, though the applicability is limited to the models where a BDD for the side constraints can be constructed. Figure 4 plots the throughput in relation to the size of each model. An inverse linear relation $tp \sim \frac{1}{\lvert\mathcal{V}\rvert}$ is clearly visible, allowing us to approximate $tp(S_{\mathcal{D}\triangleright\mathcal{F}}) \approx \frac{\alpha}{\lvert\mathcal{V}\rvert}$ with some constant α. Computing the geometric mean of the measured throughput multiplied by the number of features for each model, we get $\alpha = 1.03 \cdot 10^7$ for Zampler-RS and $\alpha = 3.9 \cdot 10^7$ for Zampler-RS-BDD, plotted as lines in Figure 4. With this estimate at hand, we can calculate the necessary hit rate which still guarantees a given effective sample rate. Rearranging Equation (1), we get a predicted *minimal hit rate* hit^* as follows:

$$hit^*(S_{\mathcal{D}\triangleright\mathcal{F}}) = \frac{esr(S_{\mathcal{D}\triangleright\mathcal{F}})}{tr(S_{\mathcal{D}\triangleright\mathcal{F}})} = esr(S_{\mathcal{D}\triangleright\mathcal{F}})\frac{\lvert\mathcal{V}\rvert}{\alpha}.$$

As an example, assume we have a feature model $\mathcal{F}$ with $\lvert\mathcal{V}\rvert = 2000$ features and want to be able to generate 1,000 samples in less than three hours. Then $esr(S_{\mathcal{D}\triangleright\mathcal{F}}) \leq \frac{1000}{3\cdot60\cdot60}$, and we get $hit(S_{\mathcal{D}\triangleright\mathcal{F}}) \geq \frac{1000\cdot2000}{3\cdot60\cdot60\cdot\alpha}$. Using Zampler-RS, the hit rate then has to be at least $1.8 \cdot 10^5$, and more than $4.7 \cdot 10^6$ for Zampler-RS-BDD. Finally, to

check if a URS-FD$^+$ approach is possible for $\mathcal{F}$, we can compute $|[\![\mathcal{F}]\!]|$ with a model counter and check if $|[\![\mathcal{F}]\!]| \geq hit^*(S_{\mathcal{D}\triangleright\mathcal{F}}) \cdot |[\![\mathcal{D}]\!]|$ holds. The hit^* columns in Table 1 give the predicted minimal hit rates to achieve an effective sample rate of at least one sample in 10 minutes, corresponding to the benchmarks in Figure 2e. The cells are colored green if $hit^* < hit(S_{\mathcal{D}\triangleright\mathcal{F}})$, and orange otherwise. The columns labeled *out* report the outcome of the prediction: true positive/negative (TP/TN) or false positive/negative (FP/FN). For almost all models, the predictions hold up. The false positives for Zampler-RS-BDD only appear in cases where the BDD constructed timed out.

Answering **RQ3**, we find that the applicability of URS-FD$^+$ approaches can be predicted from size of the feature model and the overall model count. With few side-constraints, URS-FD$^+$ is superior to diagram-agnostic URS tools.

5 Threats to Validity

Internal Validity. We mitigated performance-influencing side-effects in our experiments by running each experiment on a single core, also leaving one core unoccupied for garbage collection and maintenance threads. Further, we ran the experiments slightly longer and introduced margin on memory granted. Approaches that involve knowledge compilation and SAT solvers usually involve randomized optimizations. We hence employed state-of-the-art heuristics, e.g., relying on tree decompositions [33] for d-DNNF compilation or hypergraph cutting for BDD construction [15]. Internal risks however could be further reduced by running randomized parts of the experiments multiple times, averaging results.

External Validity. Our findings might not hold for all kinds of feature models. However, we chose from a wide range of feature models from different areas that have been shown to be representative in the field [27]. Due to the clear relation of distribution of number of features and results with only few outliers (see Figures 3 and 4) we expect our results to smoothly extend to other feature models not contained in our benchmark set. There are several methods to obtain CNF representations from feature models, possibly leading to different results also for the considered URS tools [28]. We chose the standard method well-accepted in the community and shown to be suitable in most of the cases by using FeatureIDE [38].

6 Conclusion

This paper presents the first comparative evaluation of URS approaches applied to non-trivial feature diagrams. Our findings highlight SPUR as the most effective sampler, closely followed by KUS. For feature models without side constraints, sampling directly from the feature diagram is not only theoretically superior, but also practically as demonstrated by our implementation. However, generalizing this method to handle side constraints through rejection sampling is worthwhile only for instances with very high hit rates. When the total number of valid configurations is known, we propose a method to estimate the expected hit rate and assess whether the desired number of samples can be produced within a specified time budget. In practice, we find only small feature models to have sufficiently high hit rates. For larger models commonly encountered in the literature, not enough constraints are encoded in the diagram to make diagram-based approaches viable.

References

[1] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2021. BURST: A Benchmarking Platform for Uniform Random Sampling Techniques. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*. ACM, Leicester United Kindom, 36–40. https://doi.org/10.1145/3461002.3473070

[2] Mathieu Acher, Gilles Perrouin, and Maxime Cordy. 2023. BURST: Benchmarking Uniform Random Sampling Techniques. *Science of Computer Programming* 226 (March 2023), 102914. https://doi.org/10.1016/j.scico.2022.102914

[3] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *Theory and Applications of Satisfiability Testing – SAT 2018*, Olaf Beyersdorff and Christoph M. Wintersteiger (Eds.). Vol. 10929. Springer International Publishing, Cham, 135–147. https://doi.org/10.1007/978-3-319-94144-8_9

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[5] Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the 5th Workshop on Feature-Oriented Software Development (FOSD)*. 1–8.

[6] Danilo Beuche. 2008. Modeling and Building Software Product Lines with Pure::Variants. In *2008 12th International Software Product Line Conference*. IEEE, Limerick, Ireland, 358–358. https://doi.org/10.1109/SPLC.2008.53

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. https://doi.org/10.3233/FAIA336

[8] Tomas Borovicka, Marcel Jirina Jr., Pavel Kordik, and Marcel Jirina. 2012. Selecting Representative Data Sets. In *Advances in Data Mining Knowledge Discovery and Applications*, Adem Karahoca (Ed.). IntechOpen, Rijeka, Chapter 2. https://doi.org/10.5772/50787

[9] Randal E Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[10] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41 (2003), 115–141.

[11] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Vol. 9035. Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319. https://doi.org/10.1007/978-3-662-46681-0_25

[12] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *Computer Aided Verification*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith (Eds.). Vol. 8044. Springer Berlin Heidelberg, Berlin, Heidelberg, 608–623. https://doi.org/10.1007/978-3-642-39799-8_40

[13] Paul Clements and Linda Northrop. 2001. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional.

[14] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *J. Artif. Int. Res.* 17, 1 (Sept. 2002), 229–264.

[15] Clemens Dubslaff, Nils Husung, and Nikolai Käfer. 2024. Configuring BDD Compilation Techniques for Feature Models. In *28th ACM International Systems and Software Product Line Conference*. ACM, Dommeldange Luxembourg, 209–216. https://doi.org/10.1145/3646548.3676538

[16] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 549–559. https://doi.org/10.1145/3180155.3180248

[17] Priyanka Golia, Mate Soos, Sourav Chakraborty, and Kuldeep S. Meel. 2021. Designing Samplers Is Easy: The Boon of Testers. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*. TU Wien. https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_31

[18] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, and David Benavides. 2020. Uniform and Scalable SAT-sampling for Configurable Systems. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. ACM, Montreal Quebec Canada, 1–11. https://doi.org/10.1145/3382025.3414951

[19] Ruben Heradio, David Fernandez-Amoros, José A. Galindo, David Benavides, and Don Batory. 2022. Uniform and Scalable Sampling of Highly Configurable

Systems. *Empirical Software Engineering* 27, 2 (March 2022), 44. https://doi.org/10.1007/s10664-021-10102-5

[20] Tobias Heß, Tim Jannik Schmidt, Lukas Ostheimer, Sebastian Krieter, and Thomas Thüm. 2024. UnWise: High T-Wise Coverage from Uniform Sampling. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, Bern Switzerland, 37–45. https://doi.org/10.1145/3634713.3634716

[21] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. 2024. OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Vol. 14572. Springer Nature Switzerland, Cham, 255–275. https://doi.org/10.1007/978-3-031-57256-2_13

[22] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43 (1986), 169–188. https://doi.org/10.1016/0304-3975(86)90174-X

[23] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Carnegie-Mellon University Software Engineering Institute.

[24] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. 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*. ACM, Paderborn Germany, 291–302. https://doi.org/10.1145/3106237.3106252

[25] Donald E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams* (12th ed.). Addison-Wesley Professional.

[26] Tuukka Korhonen and Matti Järvisalo. 2023. SharpSAT-TD in Model Counting Competitions 2021-2023. https://doi.org/10.48550/ARXIV.2308.15819

[27] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 898–909. https://doi.org/10.1145/3180155.3180159

[28] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or Not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. https://doi.org/10.1145/3551349.3556938

[29] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 61–71. https://doi.org/10.1145/3106237.3106273

[30] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin, Department of Computer Science.

[31] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 240–251. https://doi.org/10.1109/ICST.2019.00032

[32] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*. 139–148. https://doi.org/10.1109/RE.2006.23

[33] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 620–602. https://doi.org/10.29007/h4p9

[34] Fabio Somenzi. 2015. *CUDD: CU Decision Diagram Package*. Technical Report. University of Colorado at Boulder.

[35] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. 2020. Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Vol. 12224. Springer International Publishing, Cham, 463–484. https://doi.org/10.1007/978-3-030-53288-8_22

[36] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *28th ACM International Systems and Software Product Line Conference*. ACM, Dommeldange Luxembourg, 54–65. https://doi.org/10.1145/3646548.3672590

[37] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art # SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering* 28, 2 (March 2023), 29. https://doi.org/10.1007/s10664-022-10265-9

[38] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (Jan. 2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[39] Mathieu Vavrille, Erwan Meunier, Charlotte Truchet, and Charles Prud'Homme. 2023. *Linear Time Computation of Variation Degree and Commonalities on Feature Diagrams*. Technical Report RR-2023-01-DAPI. Nantes Uni- versité, École Centrale Nantes, IMT Atlantique, CNRS, LS2N, UMR 6004, F-44000 Nantes, France.

[40] Thomas Von Der Maßen and Horst Lichter. 2005. Determining the Variation Degree of Feature Models. In *Software Product Lines*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Henk Obbink, and Klaus Pohl (Eds.). Vol. 3714. Springer Berlin Heidelberg, Berlin, Heidelberg, 82–88. https://doi.org/10.1007/11554844_9

[41] John von Neumann. 1951. Various techniques used in connection with random digits. In *Monte Carlo Method*, A.S. Householder, G.E. Forsythe, and H.H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 36–38.

[42] Jun Yuan, K. Albin, A. Aziz, and C. Pixley. 2002. Simplifying Boolean constraint solving for random simulation-vector generation. In *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.* 123–127. https://doi.org/10.1109/ICCAD.2002.1167523

[43] Olivier Zeyen, Maxime Cordy, Gilles Perrouin, and Mathieu Acher. 2024. 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)*. ACM, Lisbon Portugal, 23–32. https://doi.org/10.1145/3644033.3644371