Scalable Sampling of Highly-Configurable Systems: Generating Random Instances of the Linux Kernel

David Fernandez-Amoros
david@issi.uned.es
Universidad Nacional de Educación a Distancia
Madrid, Spain

Ruben Heradio
rheradio@issi.uned.es
Universidad Nacional de Educación a Distancia
Madrid, Spain

Christoph Mayr-Dorn
christoph.mayr-dorn@jku.at
Johannes Kepler University
Linz, Austria

Alexander Egyed
alexander.egyed@jku.at
Johannes Kepler University
Linz, Austria

ABSTRACT

Software systems are becoming increasingly configurable. A paradigmatic example is the Linux kernel, which can be adjusted for a tremendous variety of hardware devices, from mobile phones to supercomputers, thanks to the thousands of configurable features it supports. In principle, many relevant problems on configurable systems, such as completing a partial configuration to get the system instance that consumes the least energy or optimizes any other quality attribute, could be solved through exhaustive analysis of all configurations. However, configuration spaces are typically colossal and cannot be entirely computed in practice. Alternatively, configuration samples can be analyzed to approximate the answers. Generating those samples is not trivial since features usually have inter-dependencies that constrain the configuration space. Therefore, getting a single valid configuration by chance is extremely unlikely. As a result, advanced samplers are being proposed to generate random samples at a reasonable computational cost. However, to date, no sampler can deal with highly configurable complex systems, such as the Linux kernel. This paper proposes a new sampler that does scale for those systems, based on an original theoretical approach called extensible logic groups. The sampler is compared against five other approaches. Results show our tool to be the fastest and most scalable one.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

random sampling, configurable systems, variability modeling, software product lines, SAT, binary decision diagrams, Kconfig

ACM Reference Format:

David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2022. Scalable Sampling of Highly-Configurable Systems: Generating Random Instances of the Linux Kernel. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556899

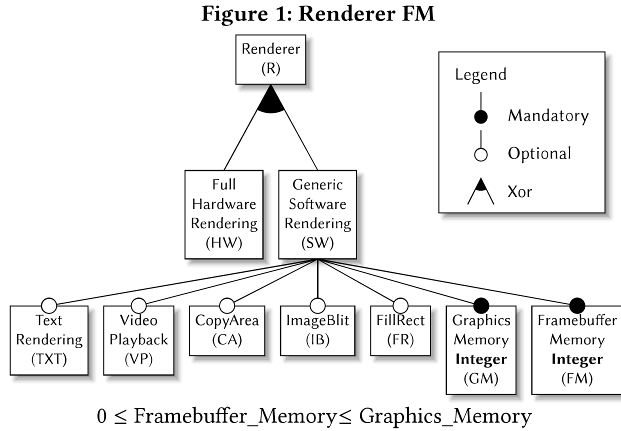
1 INTRODUCTION

The *configuration space* of most configurable systems is so vast that cannot be computed. For instance, a system from the automotive industry widely studied in the *Software Product Line* (SPL) literature encompasses $5.260 \cdot 10^{1,441}$ configurations [15]. Due to this limitation, the only way to solve many relevant problems related to those systems is by processing a configuration sample that is tractable and representative of the whole population of configurations [22, 28, 37]. Indeed, the random sampling of constrained solution spaces is not only critical for highly-configurable systems [16, 27, 30, 32–34, 42], but also for many other domains, such as software testing [11, 14, 24, 31] decision-making [17], artificial intelligence [18, 35], SPLs [16, 27, 30, 32–34, 42], hardware verification [20, 43–45] and optimization [28, 32]. However, generating such random samples is challenging, due to logical constraints between the features, which means that only a small fraction of the possible configurations are valid. For this reason, existing methods do not scale for many important highly-configurable systems, such as the Linux kernel. This paper presents a new approach that does, by breaking a complex exponential problem into a sequence of simpler, still exponential, ones.

The configurability of a system is generally specified with some type of *variability model* through a graphical or a textual notation. For example, the *Feature Model* (FM) [19] in Figure 1 visually represents the configurability of a video renderer module, which is able to provide software implementations adapted to different hardware capabilities. The parentheses indicate abbreviated feature names. The root feature is *R*, and a choice must be made between hardware rendering, *HW*, or generic software rendering, *SW*. If the latter is chosen, there are five optional features which may or may not be present in the final configuration. There are also two integer-valued features which will sometimes be omitted for clarity of exposition, a situation we will refer to as the *simplified renderer*. The hierarchical structure of FMs typically denotes that features near the top are more important than the ones at the bottom. Also, FMs often include additional cross-tree constraints, in this case, related to the amount of memory: $0 \leq$ Framebuffer_Memory$\leq$ Graphics_Memory. This model showcases some of the problems that typically arise when

performing automatic analysis of FMs, so we use it as a running example.

Figure 1: Renderer FM



0 ≤ Framebuffer_Memory≤ Graphics_Memory

Variability model analysis has evolved to the point where it is now common to also include features which allow discrete values, such as strings and numerical attributes, as opposed to simple yes/no features. Automatic analyses usually entail translating the models to Boolean logic and then analyzing the resulting formula, which is not always feasible. While Boolean features produce relatively simple formulas, discrete and numeric values result in more complicated translations. For example, in response to the SPLC'19 scalability challenge [33], Thüm et al. [39] studied sampling over Kconfig-based models. Kconfig is the textual variability notation used in the Linux kernel and other open-source projects [12]. According to Thüm et al.'s empirical results, only the smallest models could be sampled due to scalability limitations.

Part of the problem lies on *uniformity. Uniform Random Sampling* (URS) produces each of the possible configurations with equal probability [16], i.e., every member of the configuration population is equally likely to be included in a sample. Knuth, Heradio et al. [15, 21] showed that, satisfying uniformity is related to the #SAT problem, which is known to be harder than SAT [5]. Therefore, uniformity should not be taken for granted: the higher the standard for uniformity, the poorer the scalability and running time performance.

Moreover, uniform sampling is often inadequate to analyze hierarchical variability models, which are the most widely used in practice [2–4]. For example, imagine we are interested in testing the renderer system in Figure 1. The Full Hardware Rendering (*HW*) feature plays an important role, is likely to contain bugs, and it influences the performance of the resulting configuration. As only one configuration out of the 33 possible ones includes *HW*, the odds of finding this feature in a configuration sampled uniformly are 1/33 for the simplified renderer, and negligible if the numerical attributes are used (e.g., considering 32-bit integers, the probability is $\frac{1}{1+2^{36}} < 10^{-10}$). For this reason, *HW* would not be found even in a big uniform sample. As a result, alternative non-uniform sampling methods are often preferred, such as t-wise testing [13, 14, 23–26] in which a configuration sample is produced to try covering instances of all the valid combinations of a number t of features (e.g. pairs, triplets, etc.). However, even these non-uniform methods have serious scalability limitations [24].

This paper proposes a hierarchical approach to random sampling, which drops uniformity claims to achieve scalability while preserving fairness in the sense that generating a sample involves making choices, and these choices are grouped in such a way that each group gives equal weight to each possible outcome. For example, in the simplified renderer, the first choice is between *HW* and *SW*. Our fair hierarchical sampler would choose *HW* half of the time instead of 1/33 for a uniform sampler.

We present a formalism, called *Extensible Logic Groups* (ELGs), to decompose a big sampling problem into several smaller ones and sample them in sequence in such a way that each group is sampled uniformly. An implementation of the approach, named KconfigSampler is used to sample over Kconfig models. KconfigSampler performance is compared against five other state-of-the-art samplers, showing that it is the only one capable of dealing with large models, including the Linux kernel, while being faster across the board.

The remainder of this paper is organized as follows: Section 2 discusses related work about random sampling in the context of configurable systems. Section 3 introduces the theoretical concept of ELGs. Section 4 shows how to express a collection of Kconfig files as a sequence of ELGs. The results of the empirical evaluation of KconfigSampler are presented in Section 5, followed by a discussion of threats to validity in Section 6, after which we finish with our conclusions in Section 7.

2 RELATED WORK

Random sampling techniques come from two different families: those that rely on Binary Decision Diagrams (BDDs) and their extensions, and those relying on Davis-Putnam-Logemann-Loveland (DPLL)-style model counters, which will be explained later. At first glance, the scalability of the approaches seems very different across these families. Some of the techniques also take advantage of *independent support* sets [9]: *"A subset of variables whose values uniquely determine the values of the remaining variables in any satisfying assignment to the formula"*. For tools relying on independent support, the task is reduced to performing URS only on the independent support and then merely determining the value of the rest of the variables.

BDDs [6], as shown in Figure 2, are used to represent logic values as an alternative to logical formulae. A BDD is a top-down directed acyclic graph with one root. Each node has an associated variable and two children: the low child, connected with a dotted line, meaning that the corresponding Boolean variable is set to false, and the high child, connected via a straight line, meaning that the variable is set to true. It is similar to a binary decision tree in which a graph is used, instead of a tree, to reduce the number of nodes by merging any isomorphic subgraphs into one. There are also two sink nodes, called 0 and 1, which represent the logical values false and true, respectively. The variables are ordered, which means that the variable of the parent precedes that of the child in the ordering. If both children of a node are the same, the parent is omitted, which means that the variables of a parent and child need not be consecutive. The input formula to build a BDD is not required

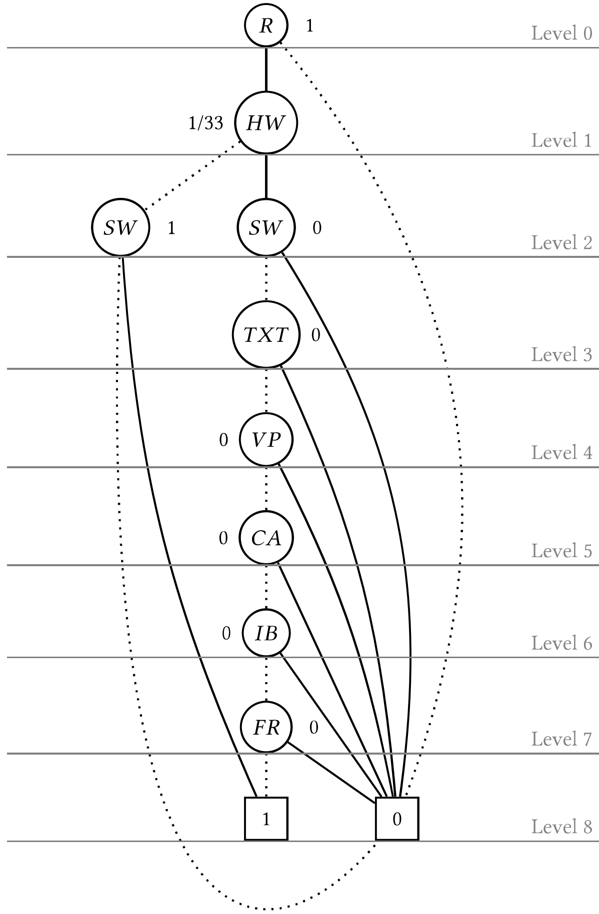


Figure 2: BDD of the simplified renderer FM

to be in any logical form, which is an advantage over DPLL methods. On the other hand, the number of nodes in a BDD is very sensitive to the variable ordering, to the point that an adequate ordering can often mean the difference between being able to build the BDD or not, and for that they have been progressively abandoned by some of the research community in favor of other structures that extend BDDs. As a conclusion, a random configuration can be found simply by traversing the graph from the root to node 1. When we follow the high child, the corresponding variable will be set to true and when the low child is followed, the variable is set to false.

DPLL [10] is the basis of a number of SAT solvers and model counters. It is customary for those to require the input to be in CNF[1], which often requires the use of techniques such as the Tseitin transformation [41] which adds up to one new artificial variable for each connective in a logical formula, plus a number of additional constraints. In its basic form, DPLL model counting consists of partially building and traversing a binary search tree, tossing variable values and simplifying the clauses with the decisions taken, and then counting the number of solutions in each branch. The whole search tree is not created because at some point in each branch either a) the remaining clauses are simple enough that the number of solutions can be counted directly or b) a conflict has been discovered and there are no solutions in the branch, which is often not detected immediately after the wrong decision has been made. The basic algorithm has been improved, among other things, with multilevel backtracking and conflict-driven clause learning, in which new clauses are learned from conflicts to speed up detecting conflicts.

We start our review of random sampling approaches with Yuan et al. [45], who used BDDs to encode circuits into logical constraints. These BDDs were then used to generate random test instances using two *procedures*: The WEIGHT procedure, in which probabilities are computed for each node, and the WALK procedure, in which the BDD is randomly traversed from the root to the true node according to the node probabilities computed before to generate a random test instance. The WEIGHT procedure relies on a set of prior probabilities for input variables, called biases, so the generation is not uniform. The approach was tested on a benchmark ranging from 76 to 1,396 variables.

The first URS algorithm for logical formulae was presented by Donald Knuth in Section 7.1.4 of [21]. Knuth described the algorithms in passing, but since they have been so influential, it is convenient to show them explicitly. The algorithms use a BDD as input. First, some properties of the BDD are computed before the generation takes place: The nodes in the BDD are traversed in reverse topological order, that is, from bottom to top, and for each node, the number of valid configurations deriving from the node is computed using that same information from its children. This information is then used to decorate each node with the probability, p, of reaching the 1 node through the high child in a random walk, as described in Algorithm 1. Figure 2 shows a BDD for the simplified renderer and the probability for each node. Generating a random configuration is as simple as traversing the BDD from the root to node 1; in each node with probability p, the next node is chosen as a Bernouilli(p) trial (i.e., the next node is the high child with probability p or the low child with probability $1 - p$). If the levels of the node and the selected child are not contiguous, the corresponding variables are assigned as Bernouilli(1/2) trials (i.e., they have the same probability of being assigned true or false) a situation described in Algorithm 2 with a function called *randomBool*. This algorithm is the same as Yuan et al.'s WALK procedure[45].

Chakraborty et al. [8] presented a tool called UniWit that used universal hashing functions to partition the search space into roughly equivalent cells. Once a cell has been decided upon, CryptoMiniSAT[2] is called to get witnesses (i.e., solutions). UniWit was evaluated against a circuit benchmark ranging from 214 to 7,624 variables. The authors mention that the benchmarks, when viewed as circuits, had up to 695 inputs, and 21 of them had more than 95 inputs each. Interestingly, they report trying to build BDDs for those 21 instances, but failed for 18 of them.

A later iteration of the tool was renamed Unigen [9] and was tested again on a hardware benchmark between 515 and 486,193 variables. The independent support ranged from 31 to 72 variables,

[1]Conjunctive Normal Form, a conjunction of clauses. A clause is a disjunction of literals. A literal is a variable or its negation.

[2]https://www.msoos.org/cryptominisat2

Algorithm 1 Node probability computation in Knuth's algorithm

```
1:  function ANNOTATE(bdd)
2:      for t ∈ ReverseLevelOrder(bdd) do
3:          if t = 0 then
4:              count(t) ← 0
5:          else if t = 1 then
6:              count(t) ← 1
7:          else
8:              thenPart ← count(t.high)2^(level(t.high)−level(t))
9:              elsePart ← count(t.low)2^(level(t.low)−level(t))
10:             count(t) ← thenPart + elsePart
11:             Pr(t) ← thenPart/count(t)
12:     return bdd
```

Algorithm 2 Knuth's random sampling algorithm

```
1:  function GENERATE(bdd)
2:      conf ← sequence of size level(1) initialized to false
3:      pos ← 0
4:      trav ← root(bdd)
5:      if trav = 0 then                              ▷ bdd represents false
6:          return conf
7:      while trav ≠ 1 do
8:          while pos < level(trav) do                ▷ Manage jumps
9:              conf[pos] ← randomBoolean()           ▷ true or false
10:             pos ← pos + 1
11:         if random() ≤ Pr(trav) then               ▷ 0 ≤ random() ≤ 1
12:             trav ← trav.high
13:             conf[pos] ← true
14:         else
15:             trav ← trav.low
16:         pos = pos + 1
17:     while pos < level(1) do                       ▷ Manage jumps
18:         conf[pos] ← randomBoolean()               ▷ true or false
19:         pos ← pos + 1
20:     return conf
```

however, which means that sampling was performed only between 31 and 72 variables. Finally, another iteration called UNIGEN2, was described in [7] and testing reached up to 777,009 variables, although the independent sets ranged from 32 to 45. The benchmark includes many instances from proprietary sources, ISCAS[3] circuits, simplified KCONFIG models and bit-blasted integer (i.e., a representation of an integer using a vector of Boolean variables representing the bits) manipulation formulas from the SMTLIB[4]. The variable sizes should be taken with a grain of salt, especially since it is customary to employ the Tseitin algorithm to obtain the CNF (which can multiply the number of variables by adding artificial variables). Consider, for instance, test example `tutorial3`: It is the biggest publicly available instance with 486,193 variables, but the minimal independent support is claimed to only have 30 variables, which means the whole model is a Boolean function of just 30 variables.

[3]https://ieee-cas.org/international-symposium-circuits-and-systems-iscas
[4]http://smtlib.cs.uiowa.edu

The implementation of UNIGEN2 complains that, if no independent support is provided, the process will be very slow.

Oh et al. [28] used *counting BDDs* to generate random samples to guide the search for near-optimal configurations. The feature models ranged from 38 to 62 features, which are already bigger than the support sets reported for UNIGEN2. The authors later mentioned that the use of counting BDDs was a limiting experimental factor. Another tool named SMARCH [29], built on top of SHARPSAT [40], counts the number of solutions and then a random number is generated to select one of those solutions. A binary search is applied to get the solution in question by determining the value of each variable successively with the aid of more calls to SHARPSAT. There is no caching of calls to SHARPSAT some of which must repeat very often. Five models derived by approximating the KCONFIG semantics of the smaller KCONFIG-based projects were analyzed. The size of the models ranged from 94 to 998 variables. The tool was used later [31] to measure t-wise testing coverage on a model with 771 variables.

QUICKSAMPLER is another sampling tool due to Dutra et al. [11]. It works by first generating a candidate solution randomly, and then applying a MAX-SAT solver to find a solution similar to the candidate. From there, a series of mutations are applied (i.e., flipping the value of some variables) to generate some more candidates until the next call to the MAX-SAT solver. QUICKSAMPLER is intended for fuzz testing, so it does not matter if some candidates are not real solutions to the constraints. QUICKSAMPLER was evaluated against the same benchmark as UNIGEN2 and it also takes advantage of independent support sets, whose reported sizes range from 17 to 481 variables.

Achlioptas et al. [1] introduced SPUR, a modification of SHARPSAT. SPUR performs a DPLL counting search like SHARPSAT, caching components for efficiency. It also stores partial solutions to produce combined solutions to the global problem, a technique called reservoir sampling. It is tested against several benchmarks varying from 14 to over 375,000 variables, but the biggest test instances are not publicly available.

The last sampling tool in this review, named KUS, was presented by Sharma et al. [36]. KUS uses a variant of BDDs, called *Deterministic Decomposable Negation Normal Form* (d-DNNF), which is a strict superset of BDDs, to perform URS. A d-DNNF is a graph consisting of AND nodes and OR nodes. Knuth's algorithms are generalized for this structure. The OR nodes represent disjunctions over disjoint variables, so the probabilities of the children can be added after some adjustment to get the probability of the parent. AND nodes also feature disjoint variable sets, so probabilities can be computed by multiplying the adjusted probabilities of the children. KUS relies on D4, a d-DNNF compiler. An advantage of KUS is that the whole sample is generated with a single traversal of the graph. KUS is evaluated successfully on a set of problems ranging from 100 to 3,979 variables, being faster than SPUR and UNIGEN2. It is a very fast algorithm, provided building the d-DNNF is viable.

We will perform a comparison of SMARCH, UNIGEN2, QUICKSAMPLER, SPUR and KUS with our own approach, KCONFIGSAMPLER, to asses their scalability and performance in Section 5. SMARCH is useful mainly as a baseline. UNIGEN2 and QUICKSAMPLER are supposed to struggle without a support set. SPUR will show the limits of optimized DPLL search and we will push the ability of

the D4 compiler, which KUS depends upon, to obtain the d-DNNF graphs.

3 EXTENSIBLE LOGIC GROUPS

All of the approaches discussed in Section 2 share the same drawback: Scalability. The problem is that the number of nodes can grow exponentially with the number of variables, meaning that it is often not possible to build the graph, for BDDs and d-DNNFs [38], or to explore the search tree for DPLL samplers. Our approach to random sampling revolves around decomposing the problem into smaller subproblems and then solving them sequentially. In particular, we will translate the FM to a conjunction of propositional formulae which we break into smaller groups.

In the following, $\mathcal{X}$ will be used to denote a set of variables, $\mathcal{F}(\mathcal{X})$ will be the set of Boolean formulae over $\mathcal{X}$ and σ will be an assignment of variables to the values $\top$ and $\bot$.

Let $(\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n)$ be a sequence of groups, such that each group $\mathcal{G}_i$ is a pair (O_i, C_i) where $\{O_i\}_{i \in \{1,2,\ldots,n\}}$ is a partition of $\mathcal{X}$, that is, $O_i \subseteq \mathcal{X}$, $\bigcup_{i=1}^{n} O_i = \mathcal{X}$ and $O_i \cap O_j = \emptyset$ if $i \neq j$, and $C_i \in \mathcal{F}(\bigcup_{j=1}^{i} O_j)$. For each group $\mathcal{G}_i$ we call $\mathcal{I}_i = \bigcup_{j=1}^{i-1} O_j$ its input variables and O_i its output variables. With this definition, each $C_i \in \mathcal{F}(\mathcal{I}_i \cup O_i)$, that is, the constraint of each group is a formula over the input and output variables in its group.

Definition 1. *A sequence of groups is called extensible, if and only if for any assignment σ_i over the variables in $\mathcal{I}_i \cup O_i$ that satisfies C_i, there is a σ_{i+1} assignment over $\mathcal{I}_{i+1} \cup O_{i+1}$ that extends σ_i and is a satisfying assignment of C_{i+1}.*

The usefulness of ELGs is that if we have a method to perform URS on constraints, then we could sample each group in an ELG consecutively and obtain a random sample of the whole problem. This sampling method would be locally uniform, but not necessarily globally uniform. Building ELGs seems easy in general, we will show how to do it for translations of Kconfig files in Section 4. In the case of the renderer feature model in Figure 1, we can decompose the problem into several groups, which we have represented in tabular fashion in Table 1. Each row of the table represents one group, explicitly showing the relevant input variables, the output variables and the constraint.

Table 1: ELGs of the Renderer FM in tabular form

Input Variables Contain	Output Variables	Constraint
	R	R
	HW, SW	(HW ∨ SW) ∧ (¬HW ∨ ¬SW)
SW	TXT	TXT → SW
SW	VP	VP → SW
SW	CA	CA → SW
SW	IB	IB → SW
SW	FR	FR → SW
SW	GM	0 ≤ GM ∧ (GM > 0 → SW)
SW, GM	FM	0 ≤ FM ∧ FM ≤ GM

In an ELG sequence, for each group $\mathcal{G}_i$, we can choose any variable in O_i as a representative of the group, which we will call the head of the group.

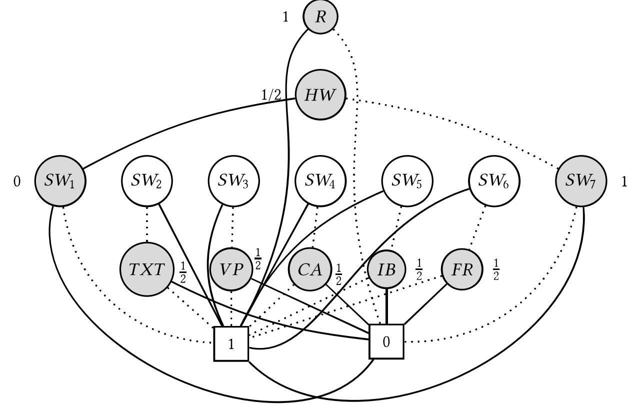
As noted earlier, we need a way to perform URS on each group in order to get a random sample. In this example, we can see that R is always true. Next we would need to choose between HW and SW. If HW is true, then TXT, VP, CA and IB are false, and GM and FM are set to zero. If SW is true, then we can choose values for all variables, provided that FM ≤ GM. When evaluating each constraint, the input variables already have a value.

We have chosen BDDs to implement URS on each group because it is a concept with solid theoretical foundations and good tool support. A forest of BDDs needs a shared variable ordering, but that comes naturally for an ELG; any ordering that places first the output variables of the first group, then the output variables of the second group and so on will do. We choose the first output variable of a group as the group head.

In this way, the input variables of a group are all the variables in the ordering preceding the head, and the output variables are the variables in the ordering from the head until the next head (not included).

We will show that scalability issues are mitigated by using groups with a small number of output variables.

Figure 3: BDD forest of the simplified Renderer FM



The idea is to generate a solution σ for the first group and then extend each such solution to the next group until we finish with all the groups. Note that when a certain group is considered, we already know the values for all its input variables. A naive approach would simplify the formulas in a group, substituting the input variables with their values, and then build a BDD and perform URS on it using Knuth's algorithms. This would be rather slow because new BDDs should be created and annotated for each group for each new solution.

However, it is better to build a BDD for each group beforehand and use them throughout the sampling process: We only need to start from the root of a BDD and traverse the graph according to the values of the input variables, which go before the output variables in the BDD ordering. When we get to the nodes corresponding to output variables, we can follow Knuth's algorithm and decide the

value according to the probability. Because the annotation algorithm works bottom-up, the probabilities are the same, regardless of the value of the input variables. In this way, we only need to annotate the forest of BDDs once, and the process should be rather efficient. As an added bonus, some of the nodes may be shared between different BDDs, reducing the overall size of the graph.

The first step, then, to perform random sampling using ELGs is to annotate the nodes with probabilities. We have carefully worded Algorithm 1 so that it can be used as is to decorate the nodes of a forest of BDDs instead of a single BDD. Figure 3 shows the result of annotating the forest of BDDs with the node probabilities for the simplified example. Input nodes (i.e., nodes corresponding to input variables) are shown in white, while output nodes are shown in gray. Input nodes have no probabilities associated because their values are already known at the time of traversal. The root nodes of the forest of BDDs to traverse sequentially are R, HW, SW_2, SW_3, SW_4, SW_5 and SW_6 and the heads are the corresponding variables.

After that, we can call Algorithm 3 to generate as many random configurations as we need. R can only be true, the probability for HW is one half and if HW is true, all the options TXT, VP, CA, IB and FR are false, otherwise, each option can be taken or not with equal probability. The difference with this approach as opposed to using only one BDD, is that all configurations are not equally likely.

Algorithm 3 Random sampling over ELGs

```
1:  function GENERATE(forest, heads)
2:  ▹ heads is a vector of the position of the first
3:  ▹ output variable for each group
4:      conf ← boolean sequence initialized to false
5:      for i ← 0 to size(heads) − 1 do
6:          trav ← roots(forest)[i]
7:          pos ← heads[i]
8:          while level(trav) < pos do        ▹ Read input variables
9:              if conf[level(trav)] then
10:                 trav ← high(trav)
11:             else
12:                 trav ← low(trav)
13:         while trav ≠ 1 do            ▹ Generate output variables
14:             while pos < level(trav) do          ▹ Manage jumps
15:                 conf[pos] ← randomBool()
16:                 pos ← pos + 1
17:             if random() < Pr(trav) then
18:                 conf[pos] = true
19:                 trav ← high(trav)
20:             else
21:                 trav ← low(trav)
22:         while pos < level(1) do                 ▹ Manage jumps
23:             conf[pos] ← randomBool()
24:             pos ← pos + 1
25:     return conf
```

An interesting property of the sampling algorithm is that the probability of each generated configuration can be computed multiplying the probabilities of the output nodes traversed in Algorithm 3: For a node with probability p, if the high child is chosen, we multiply by p, if the low child is followed, we multiply by $1 - p$.

Figure 4: Sample KCONFIG snippet

```
1  config SERIAL_CORE
2      tristate
3
4  config SERIAL_8250
5      tristate "8250/16550 and compatible serial support"
6      select SERIAL_CORE
7          ---help---
8        This selects whether you want to include the driver
         for the standard serial ports.
9
10 config SERIAL_8250_NR_UARTS
11     int "Maximum number of 8250/16550 serial ports"
12     depends on SERIAL_8250
13     default "4"
14     help
15       Set this to the number of serial ports you want the
         driver to support.
16
17 config SERIAL_8250_RUNTIME_UARTS
18     int "Number of serial ports to register at runtime"
19     depends on SERIAL_8250
20     range 0 SERIAL_8250_NR_UARTS
21     default "4"
22     help
23      Set this to the maximum number of serial ports you
        want the kernel to register at boot time.
```

This sampling method is not straightforward: Each particular problem has to be tailored to build groups conforming to the ELG definition, as we will do for the KCONFIG language in the next section in order to sample the Linux kernel, but we believe it is general enough to be useful in many situations. Digital circuits, for instance, is another prospective application field because there are clear definitions for each gate. The key is to group together the constraints in which a variable is an output value, which is why it is easier to work with intermediate level languages to delimit the groups than with the logic translations directly.

4 IMPLEMENTING ELGS FOR KCONFIG

So far, we have explained how to perform random sampling for a sequence of ELGs, but the applicability of the approach hinges on creating suitable ELGs for a particular problem. In this section, we will show how to build the logic groups for KCONFIG code. KCONFIG is a domain specific language used to configure software projects, the LINUX kernel being the original and most prominent representative. The usual workflow to get a valid configuration involves using a Kconfig interpreter application to present the user, an *application engineer*, with a series of questions about the values of the features. The final product is a configuration file which is then used as a roadmap to compile the project.

Figure 4 shows a snippet of KCONFIG code from the LINUX kernel. While we will not delve into KCONFIG's particulars[5], a few details are relevant. KCONFIG revolves around *configs*, a sort of typed variables, whose values are determined by interpreting the code and asking the user for the value of each config, possibly with some values ruled out by constraints imposed by the Kconfig code. If only one value is possible, the user is not bothered. There are five types: `bool`, `tristate`, `string`, `hex` and `int`. A typical usage of `bool` configs is to determine if a specific C source file is to be compiled or not. `Tristate` configs extend this concept allowing

[5]A more detailed explanation can be found in [12].

Figure 5: A Kconfig choice structure

```
choice
    prompt "Platform"
    default PLATFORM_LINUX

config PLATFORM_LINUX
    bool "Linux"

config PLATFORM_CYGWIN
    bool "Cygwin"

config PLATFORM_WIN32
    bool "Win32"

endchoice
```

a source file to be compiled as a loadable module. The remaining config types are mostly used to define preprocessor macros for source files. The numeric types `int` and `hex` may sporadically be involved in ordering comparisons. In the example code, it is required that the value of `SERIAL_8250_RUNTIME_UARTS` ranges from 0 to `SERIAL_8250_NR_UARTS`. Evaluating a config, c, will typically require knowing the value of other configs, which means its groups have to precede the group(s)[6] for c. It is similar, but not the same situation as the support set for Unigen2 and Quicksampler, because most configs have a prompt, meaning that the user will be asked to input the value if there is more than one acceptable value. For this reason, the input variables can restrict the admissible values for output variables, but usually not determine them. Another important construction of the Kconfig language is a *choice* between configs, which are called *members*, an example of which is shown in Figure 5. While there are several types of choices, the key is that members are mutually exclusive, i.e., only one member can be true at the same time. All the language constructions, such as depends and selects, are covered in the translation to logic.

Before we continue with the group ordering algorithm, it is convenient to explain the encoding scheme. Table 2 shows how the different types can be encoded in Boolean logic. `Bool` configs are the easiest, with one variable of the same name. `Tristate` configs require two mutually exclusive Boolean variables[7] to represent the values `yes`, `no` and `module`. `Strings` require one variable for each possible value mentioned in the Kconfig code (e.g., as a suggestion to the user or used in a comparison). A special value `OTHERVAL` is considered for strings when the user can input any value, and should be interpreted as "a value different from all the others". For `int` and `hex` values, there are two possibilities: If no ordering is involved, they are treated like `strings`. Otherwise, they are bit-blasted as 32-bit integers, to mimic the Kconfig interpreter's behavior, with one variable for sign and a master variable indicating whether the config is enabled or not. Sometimes the master or the sign are not needed, also the range of an integer is sometimes constrained so that fewer bits are necessary, which is why all the bits are marked as optional.

To obtain the variable and group ordering, we map each config to its required groups and use Algorithm 4. The idea is that the configs that do not depend on other configs values in any way (i.e.,

[6] A complex config may be encoded using several Boolean variables and several groups
[7] Meaning that both cannot be true at the same time

Table 2: Kconfig translation encoding

Type	Name	Encoding variables
boolean	X	boolean_X
tristate	X	tristate_X tristate_X_M
string	S	for each value $S_1, S_2, \ldots, S_n$ mentioned in Kconfig code, string_S_EQ_S_1 string_S_EQ_S_2 ... string_S_EQ_S_n string_S_EQ_OTHERVAL (optional)
int	N	Same as string if no ordering is involved, Bit-blasted otherwise: int_N_MASTER (optional) int_N_SIGN (optional) int_N_BIT_0 int _N_BIT_1 (optional) ... int_N_BIT_31 (optional)
hex		Same as int

they have no input variables) should go first. Then, as their values become available, they are removed from the required groups of the other configs and new configs with an empty requirements can be added the ordering and so on. Each group is represented by a group head (one of the output variables). The result of the algorithm is a group and variable order which is used for all the BDDs in the forest so that Definition 1 holds by construction. If the groups have circular requirements, the algorithm will fail. To see why, consider three groups, A, B and C, such that B needs values output in A, C needs values output in B, and A needs values output in C. There is no possible ordering of the groups. This problem can be avoided joining the three groups: the output variables are joined and the constraint is the conjunction of the constraints. The Kconfig interpreter follows a similar algorithm to derive an evaluation ordering, since the subtleties of the language often prevent evaluating configs in order of appearance: In the example, the first config, `SERIAL_CORE`, has `SERIAL_8250` in its required groups. If the user answers yes to `SERIAL_8250`, then `SERIAL_CORE` becomes true (it is selected by `SERIAL_8250`). For this reason, `SERIAL_8250` has to precede `SERIAL_CORE` in the ordering.

For the ELGs approach to succeed, choosing the right granularity is essential. Having a large number of small groups is ideal to prevent an explosion of nodes, but that might violate the ELG property. Our initial aim was to create one logic group for each config declaration in the Kconfig code and one group per choice construction.

The original setup did not work. Building a BDD with several bit-blasted integers can explode easily in the presence of ordering constraints. The reason is that BDDs have *bad memory*: Whilst the nodes in a binary search tree always remember the explicit value of the preceding variables, BDDs do not have that ability

Algorithm 4 Determining the group and variable order

```
1:  function orderGroups(groups, requiredGroups)
2:  ▷ requiredGroups maps group heads with other groups heads
3:  whose output variables it depends upon
4:      ordering ← ∅
5:      orderedHeads ← ∅
6:      changes ← ⊤
7:      while ¬empty(requiredGroups) ∧ changes do
8:          changes ← ⊥
9:          for each groupHead ∈ groups do
10:             if empty(requiredGroups(groupHead)) then
11:                 changes ← ⊤
12:                 add(outputVariables(groupHead), ordering)
13:                 add(groupHead, orderedHeads)
14:                     ▷ delete groupHead from all requiredGroups
15:                 for each otherHead ∈ groups(forest) do
16:                     delete groupHead from
17:                     requiredGroups(groups, otherHead)
18:     if ¬empty(requiredGroups(group) then
19:         No suitable order found
20:     return ordering and orderedHeads
```

because a node may have several parents, which saves nodes but loses information. If a new constraint is added to an existing BDD, such as $a \rightarrow b$, with a preceding b in the variable ordering, this requires remembering if a is true until the level of b. To recover the explicit value of a particular variable may require adding more nodes, and the further away the variable needs to be remembered, the more nodes may be added. A look at the formula for comparing two bit-blasted integers, x and y of length m, shows where the problem lies. The relation $x \leq y$ can be expressed as:

$$(x_m \rightarrow y_m) \vee (x_m \leftrightarrow y_m \wedge x_{m-1} \rightarrow y_{m-1}) \vee \ldots \vee \\ \vee (x_m \leftrightarrow y_m \wedge x_{m-1} \leftrightarrow y_{m-1} \wedge \ldots \wedge x_0 \leftrightarrow y_0)$$

Each bit is always compared with its equally significant counterpart. That is why it is very convenient to have the integers aligned, so that all the master variables are consecutive, all the sign variables are together and every i^{th} bit are also grouped together in the variable ordering. In the Kconfig snippet, we can see that `SERIAL_8250_RUNTIME_UARTS` depends on `SERIAL_8250_NR_UARTS`, so all the variables of `SERIAL_8250_NR_UARTS` precede those of `SERIAL_8250_RUNTIME_UARTS`. The separation between concerned bits is 32 variables long. Since it is not possible to build the BDD for `SERIAL_8250_RUNTIME_UARTS` under these conditions, our first reaction was to separate each bit variable in its own group, so that each bit of `SERIAL_8250_RUNTIME_UARTS` could use its counterpart in `SERIAL_8250_NR_UARTS` as an input variable. Sadly, we discovered that, in the general case, circular requirements can appear if there are other config declarations in between the numeric configs being compared. In that case, Algorithm 4 would never end because no suitable ordering can be found. Finally, we resorted to joining groups involved in ordering constraints and the groups in between into only one group, with the bits aligned.

We got a little further building the BDDs in this way, but this strategy also failed. Some config definitions of type `string` (also `int` and `hex` when encoded as strings) are too complicated and the BDDs still blow up. When there is no prompt for the user to input a value (i.e., the output variables are fully determined by the input variables), it is possible to split the original groups so that each individual string value is the output variable of its own group. We tried to build the forest of BDDs with this finer-grained setup, only to find it still blew up.

The solution was to add some constraints already present in other groups. If two mutually exclusive input variables appear in a group, it is very convenient to include that information in the group, even if it is already present in their defining groups, to keep the BDD size under control. There are two important sources of such exclusive variables: `string` values are mutually exclusive (i.e., a string cannot have two different values) as well as `choice` members. Enriching the groups with this information reduced the number of nodes to a manageable degree, as we will show in Section 5.

5 EXPERIMENTAL RESULTS

5.1 Research questions

We have structured the evaluation around two *Research Questions*:

- **RQ1: Feasibility and scalability of encoding configuration models as ELGs.** Is it possible to synthesize ELGs for real problems in a scalable way, e.g., for the Linux kernel?
- **RQ2: Sampling performance comparison.** How well does ELG-based sampling compare to other existing samplers?

5.2 Experimental setup

To evaluate the competing approaches, we have created a new translation of Kconfig projects to Boolean logic. While the details are outside the scope of this paper, the translation is available at the following public repository, which also includes all the information and tools necessary to replicate our experiments:

https://github.com/davidfa71/Sampling-the-Linux-kernel

To the best of our knowledge, our translation is the first one that is complete: All language constructions have been translated for all variable types, including environment variables and numerical ordering constraints, with an emphasis on simplifying the formulas as much as possible. The translated systems are summarised in Table 3. These projects are Kconfig veterans and thus commonly used in the literature. Toybox and BusyBox are implementations of common UNIX commands in a single executable, meant for embedded systems, axTLS is an open source implementation of the TLS protocol, μClibc is an minimal implementation of the C library, EmbToolKit is a toolkit for embedded systems, which includes BusyBox. Coreboot is an alternative to BIOS and deals mainly with motherboards. Fiasco is a real-time microkernel. Freetz is a toolbox for router firmware. BuildRoot is a cross-compilation tool to create embedded Linux kernels.

The Linux kernel makes heavy use of `tristate` configs, which explains the big difference between the number of `configs` (i.e., features) and the number of Boolean variables in the translation.

Table 3: Tested systems

Model	Version	#Features	#Variables
Fiasco	2021.04.30	78	98
axTLS	2.1.15	94	119
ToyBox	2021.08.21	340	342
μClibc	0.9.33	342	386
BusyBox	1.33.0	1,050	1,370
EmbToolKit	1.9.0	2,824	3,392
CoreBoot	4.13	4,460	5,604
Freetz	2021.06.15	4,870	6,172
BuildRoot	2021.02.01	7,213	8,106
Linux x86	3.9.4	8,670	15,362

The experiments were carried out on a HP Proliant server with two Xeon E5-2660v4 processors with 28 cores each and 224 Gb of memory. To manage the BDDs, we used the CUDD[8] library, which has built-in support for BDD forests.

5.3 ELGs feasibility and scalability (RQ1)

As explained in Section 3, our goal is to perform random sampling using a previously synthesized BDD forest for the feature model in question. In Section 4 we explained how to apply the ELGs approach to a translation of Kconfig projects. Table 4 shows the results of building these forests.

Table 4: Synthesis of the BDD forests

Model	#Nodes	#Groups	Time (sec)
Fiasco	188	77	0.07
axTLS	238	80	0.08
ToyBox	118	338	0.31
μClibc	799	251	0.37
BusyBox	2,260	1,034	2.70
EmbToolKit	37,091	2,839	17.15
CoreBoot	236,248	3,696	88.77
Freetz	137,241	5,036	130.62
BuildRoot	416,987	7,042	106.02
Linux x86	422,109	8,864	229.29

We managed to successfully build the forest of BDDs for all the evaluated projects in a reasonable amount of time. The tuning of the ELGs has paid off to the point where even the Linux kernel fits in less than half a million nodes, which is surprising given the complexity of this model. All the information is there, so it can be considered as a precursor to a *knowledge compilation representation* of the kernel. Of course, building one single BDD for the whole kernel still seems a distant possibility, but it is just a matter of performing an `AND` operation on the forest roots.

All the information about a forest can be adequately stored in a file format called `DDDMP`, which interoperates with CUDD and has fields to store the variable and root names in order, so that the sampling algorithm can determine which are the input and output variables for each BDD. The files are included in the repository.

Table 5 shows the top config contributors (with names abbreviated) to the number of nodes in the BDD forest. It is interesting to note that the distribution of node numbers per config is heavily skewed. The vast majority of features translate to a very simple BDD. There are notable exceptions, though: a single config in CoreBoot accounts for 78% of the nodes. Three configs account for 76% of the nodes in Freetz and 83% of the nodes in BuildRoot. For Linux, two configs make up 73% of the nodes and removing just four configs would leave the BDD forest with less than 100,000 nodes. These hot spots give clear indications of where to look to improve the translation and the building of the BDDs in future work.

5.4 Sampling performance comparison (RQ2)

In Experiment 1, we set up a number of competing samplers [1, 7, 11, 29, 36] and our own KconfigSampler to produce a sample with 1,000 configurations within a timeout of 24 hours. Our ELG-based sampler and encoder, KconfigSampler, not only samples the variables, but also decodes the Boolean variables into the corresponding Kconfig types (`string`, `int` and so on), so that the result is a sample of legal configurations, as far as Kconfig is concerned, ready to compile. To level the playing field, we instructed our approach to merely sample the Boolean variables and skip decoding. Table 6 summarizes the results. The last column shows the results of KconfigSampler. We show total running times. No averages or box-plots were deemed necessary because the sample size is big enough. Smarch's less-than-stellar performance was predictable, since it issues a call to sharpSAT for each variable. Starting from BusyBox, sharpSAT is unable to even count the number of solutions, the first step of the algorithm. QuickSampler is indeed faster than Unigen2, but both suffer greatly from the change of context and the lack of support sets. In Kconfig systems, the vast majority of configs include a prompt, i.e., a way to ask the user for the value of a feature with some values possibly excluded (which makes sense since the whole intent is to help the application engineer configure the system). For this reason, the independent support would be almost as big as the whole set of variables. This is in stark contrast to circuits, where a few inputs are incorporated into a much bigger number of logic gates. In any case, Unigen2 quickly shows its scalability issues, with QuickSampler following a couple of tests later. SPUR and KUS do the best work among the alternative approaches, with SPUR scaling up even to CoreBoot. KUS fails when the d4 d-DNNF compiler does. Building a single d-DNNF graph of the bigger systems is somewhat similar to building a single BDD for them, a feat that keeps eluding the SPL community. Our approach is the only one to scale up to all the systems and it beats every other except for ToyBox, in which SPUR was slightly faster.

We felt that Experiment 1 showed our approach in an unfavorable light. Because the first step is computing the node probabilities, which does not depend on sample size, a small size makes the approach look slower. The sample size was dictated by the shortcomings of the other approaches rather than the merits of ours, so we tested it again to build a sample of size one million. The results in Table 7 show that the approach has a strong performance

[8] https://github.com/ivmai/cudd

Model	Config	#Nodes	%
Fiasco	SCHED_PIT	22	11.70
	MP_MAX_CPUS	19	10.10
	PF_PC	8	4.25
	SCHED_FIXED_PRIO	7	3.73
	PERF_CNT	6	3.19
axTLS	SSL_SERVER_ONLY	9	3.78
	JAVA_HOME	7	2.94
	HTTP_WEBROOT	7	2.94
	X509_MAX_CA_CERTS	7	2.94
	VISUAL_STUDIO_7_0	7	2.94
ToyBox	LSM_NONE	5	4.23
	TEST_GLUE	4	3.38
	MKNOD_Z	4	3.38
	MKFIFO_Z	4	3.38
	MKDIR_Z	4	3.38
μClibc	TARGET_alpha	55	6.88
	TARGET_ARCH_EQ_OTHERVAL	55	6.88
	DEPRECATED_SYSCALLS	27	3.30
	UCLIBC_FORMAT_FDPIC_ELF	22	2.75
	UCLIBC_PWD_BUFFER_SIZE	20	2.50
BusyBox	LAST_ID	371	16.41
	VI_UNDO_QUEUE_MAX	48	2.12
	BEEP_FREQ	48	2.12
	SYSLOGD_READ_BUFFER_SIZE	40	1.76
	VI_MAX_LEN	39	1.72
EmbTool Kit	ARCH_ARM_FPU_VFP	24,588	66.29
	BUSYB_PLATFORM_LINUX	267	0.71
	NLS_STRING_EQ_zh_TW	103	0.27
	NLS_STRING_EQ_zh_HK	102	0.27
	NLS_STRING_EQ_zh_CN	101	0.27
CoreBoot	SMBIOS_EQ_986LCD-M	185,953	78.71
	VGA_EQ_1002,1304	6,926	2.90
	CBFS_SIZE_EQ_0x00040000	6,500	2.75
	VGA_EQ_PicassoVbios.bin	4,327	1.83
	SMBIOS_EQ_51NB	4,073	1.70
Freetz	DL_SOURCE	38,707	28.20
	DL_SOURCE_MD5	38,621	28.14
	DL_SITE	27,839	20.28
	TYPE_FIRMWARE_04_XX	3,072	2.23
	DL_KERNEL_SOURCE_MD5	2,814	2.05
Build Root	ARCH_SUPPORTS	171,519	41.13
	BLEEDING_EDGE	128,769	30.88
	PACKAGE_OPENSSL	46,970	11.26
	PACKAGE_ZLIB	13,448	3.22
	PACKAGE_LIBGLIB2	9,875	2.36
Linux x86	INPUT_FF_MEMLESS	249,951	59.21
	FB_DDC	59,385	14.06
	USB_ZERO	10,458	2.47
	FW_LOADER	3,930	0.93
	SND_PCM	2,667	0.63

Table 5: Biggest contributors to forest of BDDs

and can deliver one million configurations in less than an hour, even for the Linux FM. The other systems are too slow to repeat Experiment 1 for one million configurations but we did it only for ToyBox with SPUR, and it took 162 seconds, i.e., four times longer than our approach, thus confirming our suspicion.

A final reflection concerns the use of CNF. Our translation is not in CNF and the competing tools need it, so we had to convert it. We tried the direct approach, but the transformation grew exponentially for the bigger models, leaving us with the Tseitin transformation [41] to contain such growth. Table 8 shows the difference in variables before and after the transformation, and also the growth from formula fragments into clauses. CNF is a *de facto* standard in the SAT community, but it clearly affects the performance of the tools. We tried to minimize the number of artificial variables added but the table shows that the effect is very noticeable in the largest systems, precisely those for which the alternative approaches failed. The relation between these two facts is intriguing and deserves more research as it may test the limits of what can be achieved with state-of-the-art SAT technology.

6 THREATS TO VALIDITY

The main threat to validity concerns the practical applicability and scalability of the approach. As we have shown, groups with a small number of variables will produce smaller BDDs, but if the groups are too small, Algorithm 4 may fail to produce a group ordering because of circular requirements. This problem showed up with the range construction in Kconfig which translates to integer ordering constraints, and we solved it automatically by joining all the intervening groups, a simple but effective strategy. In some cases, a group may be created that is too big for the BDD building to succeed. Splitting a group into several ones is also possible in some cases to keep the BDDs at bay. Supplementing the group constraints with additional constraints between the input variables also helped keeping the BDDs from growing too much. We feel confident that KconfigSampler will easily handle Kconfig projects for the foreseeable future. In the general domain, crafting the groups with the right granularity (i.e., big enough to avoid circularity but small enough to be able to build BDDs) might be hard, which is why some knowledge of the underlying semantics is so useful: One group per definition is a good starting point.

Another aspect of the approach is the use of BDDs. ELGs can be used with any sampling procedure, so if building BDDs should become a limiting factor, they might be exchanged for any other sampling method, including those evaluated in Section 5.

7 CONCLUSIONS

In this paper, we have shown how traditional approaches to random sampling of constrained configuration spaces fail to scale to large highly-configurable systems, specifically in the case of Kconfig variability models, and have provided a new approach based on extensible logic groups, which are better suited for sampling hierarchical models, in which the upper levels are more important than the lower ones. While deriving these groups from a problem is not straightforward, we believe that the approach is general enough to cover many interesting problems in other domains, including the hardware testing problems upon which URS is usually tested.

Table 6: Experiment 1: Sampling time in seconds for one thousand configurations

Model	SMARCH [29]	UNIGEN2 [7]	QUICK SAMPLER [11]	SPUR [1]	KUS[36]	KCONFIG SAMPLER
FIASCO	6,699	7.94	0.377	0.104	0.315	0.039
AXTLS	7,268	45.89	0.421	0.205	0.450	0.054
TOYBOX	7,366	185.62	0.103	0.059	0.158	0.082
μCLIBC	52,740	<timeout>	3.007	7.866	1.079	0.167
BUSYBOX	<timeout>	<timeout>	13.07	365.56	6.158	0.305
EMBTOOLKIT	<timeout>	<timeout>	<timeout>	<timeout>	<D4 timeout>	1.322
COREBOOT	<timeout>	<timeout>	<timeout>	2164.9	<D4 timeout>	2.650
FREETZ	<timeout>	<timeout>	<timeout>	<timeout>	<D4 timeout>	2.389
BUILDROOT	<timeout>	<timeout>	<timeout>	<timeout>	<D4 timeout>	4.264
LINUX x86	<timeout>	<timeout>	<timeout>	<timeout>	<D4 timeout>	7.797

Table 7: Experiment 2: Time KCONFIGSAMPLER took to generate one million configurations

Model	Time (sec)
FIASCO	17.079
AXTLS	19.801
TOYBOX	40.853
μCLIBC	60.688
BUSYBOX	200.520
EMBTOOLKIT	703.514
COREBOOT	1,332.406
FREETZ	1,305.746
BUILDROOT	1,965.860
LINUX x86	3,447.617

Table 8: Growth due to Tseitin transformation

Model	#Vars. Bef	#Vars. After	#Frags.	#Clauses
FIASCO	98	306 ×3.12	135	710 ×5.25
AXTLS	119	374 ×3.14	207	920 ×4.44
TOYBOX	342	411 ×1.20	85	282 ×3.31
μCLIBC	386	1,214 ×3.14	1,378	3,687 ×2.67
BUSYBOX	1,370	4,153 ×3.03	795	8,579 ×10.79
EMBTOOLKIT	3,392	31,181 ×9.19	6,430	78,545 ×12.21
COREBOOT	5,604	69,836 ×12.38	165,609	345,749 ×2.08
FREETZ	6,172	67,546 ×10.94	63,394	240,767 ×3.79
BUILDROOT	8,106	54,080 ×6.67	62,290	194,017 ×3.11
LINUX x86	15,362	186,059 ×12.11	35,919	527,240 ×14.67

We supported this claim by customizing the technique to the variability models created from KCONFIG files for ten open-source projects, including the Linux kernel, and gathered important insights in the process:

- As the approach hinges on the synthesis of a BDD for each extensible group, the granularity of the groups is of paramount importance.
- Bit-blasted integers involved in ordering comparisons are best managed when they belong to a single group and variable ordering keeps the bits aligned by their level of significance.
- The more complex config types (strings and numbers without a prompt), which translated into several Boolean variables, needed breaking down into one group for each discrete value.
- All groups were reinforced when possible with constraints regarding mutually exclusive input variables to avoid BDD building from blowing up.

We managed to create a forest of BDDs for each tested project with a very manageable footprint of less than half a million nodes in all cases, including the LINUX kernel. The experimental section pointed out some possible areas of improvement with respect to BDD size. We also compared the performance of the approach to five other alternative tools [1, 7, 11, 29, 36], with our approach being faster and more scalable than all the competitors, often by several orders of magnitude. The approach can generate a million sample configurations for the LINUX kernel in less than an hour.

A closer examination of the test cases revealed a consistent trend regarding the use of the Tseitin construction to deliver the CNF format required for the other approaches, namely that it produces an explosive growth in the number of both variables and clauses which severely affects their scalability.

Finally, all the logical models and our sampling tool, KCONFIGSAMPLER, have been made publicly available to the community for further research.

ACKNOWLEDGMENTS

This work has been supported by the Universidad Nacional de Educacion a Distancia (project references 2021V/PUNED/008 and 2022V/PUNED/007).

REFERENCES

[1] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Oxford, UK, 135–147.

[2] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases of Feature-Based Variability Modeling

in Industry. In *Model-Driven Engineering Languages and Systems International Conference (MODELS)*. Valencia, Spain, 302–319.

[3] Thorsten Berger, Steven She, R. Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[4] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2020. The state of adoption and the challenges of systematic variability management in industry. *Empirical Software Engineering* 25, 3 (2020), 1755–1797.

[5] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press.

[6] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691.

[7] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. London, UK, 304–319.

[8] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *25th International Conference on Computer Aided Verification (CAV)*. Saint Petersburg, Russia, 608–623.

[9] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2014. Balancing Scalability and Uniformity in SAT Witness Generator. In *51st Annual Design Automation Conference (DAC)*. San Francisco, CA, USA, 1–6.

[10] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

[11] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden, 549–559.

[12] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A Kconfig translation to logic with one-way validation system. In *23rd International Systems and Software Product Line Conference (SPLC)*. Paris, France, 303–308.

[13] Michael Forbes, James Lawrence, Yu Lei, Raghu Kacker, and D. Kuhn. 2008. Refining the In-Parameter-Order Strategy for Constructing Covering Arrays. *Journal of Research of the National Institute of Standards and Technology* 113, 5 (2008), 287–297.

[14] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* 24, 2 (2019), 674–717.

[15] Ruben Heradio, David Fernandez-Amoros, JoséA. Galindo, David Benavides, and Don Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* 27, 2 (2022), 44.

[16] Ruben Heradio, David Fernandez-Amoros, Jose A. Galindo, and David Benavides. 2020. Uniform and scalable SAT-sampling for configurable systems. In *24th Systems and Software Product Line Conference (SPLC)*. Montréal, Canada, 1–11.

[17] Jose Miguel Horcas, José A. Galindo, Ruben Heradio, David Fernandez-Amoros, and David Benavides. 2021. Monte Carlo tree search for feature model analyses: a general framework for decision-making. In *25th ACM International Systems and Software Product Line Conference*, Vol. A. Leicester, UK, 190–201.

[18] Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. 2016. On computing minimal independent support and its applications to sampling and counting. *Constraints* 21, 1 (2016), 41–58.

[19] 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 CMU/SEI-90-TR-021. Software Engineering Institute.

[20] Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus generation for constrained random simulation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. San Jose, CA, USA, 258–265.

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

[22] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling* 18, 3 (2019), 2265–2283.

[23] Sebastian Krieter. 2019. Enabling Efficient Automated Configuration Generation and Management. In *23rd International Systems and Software Product Line Conference (SPLC)*. Paris, France, 215–221.

[24] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *14th International Working Conference on Variability Modelling of Software-Intensive System (VaMoS)*. Magdeburg, Germany, 1–10.

[25] Yu Lei, Raghu Kacker, D. Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing. *Software Testing Verification & Reliability* 18, 3 (2008), 125–148.

[26] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 549–556.

[27] Daniel-Jesus Muñoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *23rd International Systems and Software Product Line Conference (SPLC)*. Paris, France, 289–301.

[28] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany, 61–71.

[29] Jeho Oh, Don S. Batory, Marijn J. H. Heule, Margaret Myers, and Paul Gazzillo. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report TR-19-02. Department of Computer Science. The University of Texas at Austin.

[30] Jeho Oh, Paul Gazillo, Don Batory, Marijn J. H. Huele, and Margaret Meyers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin.

[31] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *23rd International Systems and Software Product Line Conference (SPLC)*. Paris, France, 84–87.

[32] Jeho Oh, Margaret Myers, and Don Batory. 2016. *Finding Product Line Configurations with High Performance by Random Sampling*. Technical Report TR-16-22. Department of Computer Science, University of Texas at Austin.

[33] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *23rd International Systems and Software Product Line Conference (SPLC)*. Paris, France, 78–83.

[34] 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 *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Xian, China, China, 240–251.

[35] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, Florida, USA, 224–234.

[36] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation meets Uniform Sampling. In *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Awassa, Ethiopia, 620–636.

[37] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 284–294.

[38] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. Magdeburg, Germany.

[39] Thomas Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *24th ACM Conference on Systems and Software Product Lines (SPLC) (SPLC '20)*. Montreal, Quebec, Canada, 1–6.

[40] Marc Thurley. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Seattle, WA, USA, 424–429.

[41] Gregory S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, 466–483.

[42] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *22md International Systems and Software Product Line Conference (SPLC)*. Gothenburg, Sweden, 1–13.

[43] Jun Yuan, Ken Albin, Adnan Aziz, and Carl Pixley. 2002. Simplifying Boolean Constraint Solving for Random Simulation-Vector Generation. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. San Jose, CA, USA, 123–127.

[44] Jun Yuan, Ken Albin, Adnan Aziz, and Carl Pixley. 2002. Simplifying Constraint Solving in Random Simulation Generation. In *11th IEEE/ACM International Workshop on Logic & Synthesis (IWLS)*. New Orleans, Louisiana, USA, 185–190.

[45] Jun Yuan, Kurt Shultz, and Carl Pixley. 1999. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *International Conference on Computer-Aided Design (ICCAD)*. San Jose, CA, USA, 584–589.

View publication stats