Uniform Sampling from Kconfig Feature Models

Jeho Oh, Don Batory, Marijn Heule, Margaret Myers
University of Texas at Austin
Department of Computer Science
Texas, USA

Paul Gazzillo
University of Central Florida
Department of Computer Science
Florida, USA

***Abstract*—Random sampling of configuration spaces is a useful tool for working with software product lines (SPLs), enabling, analyzing and reasoning about spaces too large for exhaustive exploration. Being able to create uniform sampling is critical for making statistical inferences about SPLs, but it particularly hard for massive, real-world systems. We show how uniform random sampling can be done on systems that use the Kconfig feature modeling language, a popular choice among low-level and embedded systems. Despite its importance, prior work considered uniform random sampling infeasible and sampled configurations without confirming that their samples were uniformly distributed. Especially, existing Kconfig models were not known to be suitable for uniform random sampling, although they were used extensively. We solve these challenges with algorithmic advances embodied in two tools: Kclause to produce compact models and Smarch to perform efficient sampling guaranteed to be uniform. Kclause has been extended to model Kconfig semantics with hand-tuned, simplified formulas that minimizing the number of clauses, a critical property for fast sampling. Smarch is a new recursive, sampling algorithm based on sharpSAT that supports massive configuration spaces with 38% faster sampling times.**

I. Introduction

Software Product Lines (SPLs) are highly-configurable systems. A *feature* is an increment in system functionality and a variability among system products. Each legal combination of features is a *configuration*. The set of all configurations is a *configuration space* [1].

Kconfig is a variability management tool that is extensively used for open-source, real-world SPLs [2]. A *Kconfig system* allows its users to build a customized version by selecting desired features (functionalities and environments), where its variability options are defined in *Kconfig models*.

Kconfig models have huge numbers of features and complex constraints. It is not unusual for a Kconfig model to have a configuration space whose size exceeds 10^{82}, the number of atoms in the Universe [3]. It comes as no surprise that automated analyses of Kconfig models are needed. Automated solvers, like SAT [4] and #SAT [5], can check conflicts in constraints among features, derive valid configurations, and perform diverse analyses such as: understand the structure of a system [6]–[12], find variability bugs [13]–[15], detect dead features and inconsistencies between code and features [16]–[19], and perform multi-objective optimizations [20]–[23].

Uniform Random Sampling (URS) is relevant to these analyses as it is a basic way to derive unbiased statistics about a configuration space. URS was required in prior work to derive the influence of a feature for performance modeling [24], [25], perform multi-objective optimization [20]–[23], and evaluate different sampling approaches to locate variability bugs [13], [15].

As late as 2016, URS of SPL configuration spaces was considered infeasible [15], [19]. In 2017, Oh *et al.* [26] showed how URS could be performed; their techniques relied on BDDs [27], which limited the scope of their tool. A goal of this paper is to scale [26] to URS spaces of colossal size. In doing so, we discovered two technical requirements for URS that are generally not satisfied by existing tools — *especially* those that translate Kconfig models:

1) CNF formulas must not encode configurations redundantly. When URS relies on counting the number of solutions of a CNF formula, redundant configurations can bias statistics about a configuration space, and
2) Large CNF formulas have large sampling times; CNF formulas should have as few variables and clauses as possible.

We address these problems with a pair of tools, Kclause and Smarch, to analyze Kconfig models. Their novelties are:

- Kclause translates a Kconfig model into a CNF formula that does not represent configurations redundantly;
- Kclause formulas are compact, having 63% fewer clauses and 44% faster sampling time on average than formulas produced by other tools;
- Smarch can URS colossal configuration spaces $\gg 10^{82}$ in size — no other tool can come close;
- Experimental results are demonstrated using Kclause and Smarch on Kconfig systems. One system (*Busybox*) has a titanic space of 10^{248} products; and

II. Background

A. Feature Model

Every SPL has a feature model that defines all SPL features and feature constraints. A typical constraint says selecting a feature requires or inhibits the selection of other features. For a feature model μ, $\mathcal{C}(\mu)$ is its configuration space and $|\mathcal{C}(\mu)|$ or simply $|\mu|$ is its size — the number of configurations.

Every classical feature model μ can be converted into a propositional formula $\phi(\mu)$, or simply ϕ, whose variables are SPL features and whose solutions are configurations [1].[1] Automated solvers can analyze ϕ to find configurations using

[1]Non-classical feature models have numerical and textual constraints [28]. We deal with classical feature models in this paper [1].

satisfiability (SAT) solvers and to count the number of configurations using *#SAT* solvers [29]. Identities in this universe are:

$$\mathcal{C}(\mu) \equiv \mathcal{C}(\phi) \quad (1)$$

$$|\mathcal{C}(\mu)| = |\mathcal{C}(\phi)| \quad (2)$$

Eq. (1) says each legal configuration in μ is a unique solution of ϕ and vice versa. Eq. (2) says the number of legal configurations in μ equals the number of solutions of ϕ.

B. Uniform Random Sampling

Chakraborty *et al.* [30] were the first to create a tool to sample solutions of large propositional formulas. They used an approximate #SAT solver [31], a technology quite different from what we use. Their tool is Unigen2 which we examine further in Sec. V.

Oh *et al.* [26] were the first to demonstrate URS of a SPL configuration space. Assuming Eq. (2), a uniform random number generator selects an integer j in the range $[1..|\phi|]$ while a *Counting Binary Decision Diagram* (CBBD) [27] encodes all possible solutions of ϕ in a BDD, along with the number of solutions in each subtree. j was converted into a configuration by walking the CBBD top-down using binary digits of j as a traversal guide. Thus, a URS of range $[1..|\phi|]$ yields a URS of $\mathcal{C}(\phi)$. CBBDs, however, have well-known scalability problems and are incapable of sampling colossal spaces of Kconfig models. Our development of Smarch, discussed later, replaces CBBDs with #SAT tools that scale much better.

C. CNF Formula Conversion

SAT and #SAT solvers require a *Conjunctive Normal Form* (CNF) formula as input [32], [33]. Transforming ϕ into a CNF formula ϕ^{cnf} is straightforward using rules of logical equivalence. But doing so may increase the number of clauses exponentially [32] and simplifying ϕ^{cnf} to reduce the number of clauses is nontrivial [34], [35].

To avoid such problems, *equisatisfiable transformations* (ETs) are used. ETs produce a CNF formula ϕ_{es}^{cnf} that is equisatisfiable to ϕ [36]. Two formulas are *equisatifiable* when one formula is satisfiable only if the other is satisfiable. There are a many ETs [36]–[38] not all of which are suitable for URS. Consider:

$$\phi = (a \wedge b) \vee (c \wedge d)$$

An ET from Plaisted & Greenbaum [38] introduces additional variables x_1 and x_2 for the clauses of ϕ:

$$\phi_{es}^{cnf} = (x_1 \vee x_2) \wedge (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg x_2 \vee c) \wedge (\neg x_2 \vee d)$$

Each row of Table I is a solution of both ϕ and ϕ_{es}^{cnf}. The last solution of ϕ corresponds to 3 solutions of ϕ_{es}^{cnf}. A problem addressed in this paper is now exposed: URS of $\mathcal{C}(\phi_{es}^{cnf})$ is a ***biased*** sampling of $\mathcal{C}(\phi)$. Statistical predictions by URS of $\mathcal{C}(\phi_{es}^{cnf})$ are distorted predictions about $\mathcal{C}(\phi)$. Consider:

- $|\mathcal{C}(\phi_{es}^{cnf})|$ is 9 and $|\mathcal{C}(\phi)|$ is 7, a 28% over-estimation.
- The percentage of products with feature d in $\mathcal{C}(\phi_{es}^{cnf})$ is $78\% = \frac{7}{9}$, whereas the correct answer in $\mathcal{C}(\phi)$ is $71\% = \frac{5}{7}$, a 10% over-estimation.

ϕ				ϕ_{es}^{cnf}					
a	b	c	d	a	b	c	d	x_1	x_2
1	1	0	0	1	1	0	0	1	0
1	1	0	1	1	1	0	1	1	0
1	1	1	0	1	1	1	0	1	0
0	0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0	1
1	0	1	1	1	0	1	1	0	1
1	1	1	1	1	1	1	1	1	0
				1	1	1	1	0	1
				1	1	1	1	1	1

TABLE I: Solution Comparison Between ϕ and ϕ_{es}^{cnf}

So how do redundant solutions arise? We discovered that if ϕ_{es}^{cnf} adds *no* new variables to ϕ then we are OK: URS statistics about $\mathcal{C}(\phi_{es}^{cnf})$ will match $\mathcal{C}(\phi)$ because $|\mathcal{C}(\phi_{es}^{cnf})| = |\mathcal{C}(\phi)|$.

Adding variables *might not be* a problem. Tseitin's transformation [36], a well-known ET method, adds variables but does *not* increase the number of solutions. Tseitin's transformation extends the Plaisted & Greenbaum transformation with so-called blocked clauses [39]. However, the elimination of blocked clauses [40], which is a SAT preprocessing technique used in top-tier solvers, removes those clauses and introduces redundant solutions.

The example of Table I shows there are 'bad' ETs that both add variables *and* distort statistical predictions. The problem is this: ***Given a feature-model-to-propositional-formula tool, you do not know if it uses 'bad' ETs if it uses extra variables.***

Our Kclause tool of Sec. III avoids the above controversies because it adds no extra variables.

D. Kconfig

Kconfig is a tool for managing the variability of a system and deriving valid configurations. It was originally developed for the Linux kernel, but also has been used in embedded libraries, client-server frameworks, and firmware infrastructures [9]. Kconfig systems are widely used today to evaluate SPL ideas due to their large number of features, nontrivial constraints, and availability as open source. Fig. 1 is a rough overview of how Kconfig configures a system.

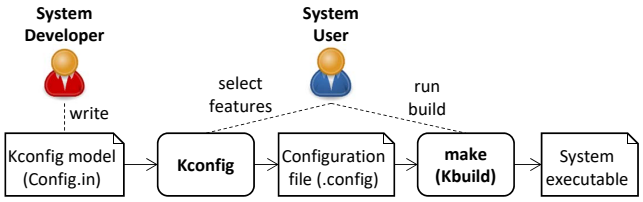

Fig. 1: Overview of Kconfig Usage.

Kconfig manages variability at build-time. A system developer specifies features and their constraints in a Kconfig model. Kconfig takes this model as input and allows users to select or deselect features accordingly. Feature selection can be done manually (menuconfig) or automatically using Kconfig functionalities such as defconfig, randconfig,

and allyesconfig [2].[2] The output of Kconfig is a *configuration file* that lists each feature and if it is selected or not. A configuration file is used to build the target system. The build tool reads the configuration file and determines the portions of source files to compile.

A Kconfig model is a SPL feature model. A Kconfig model can be merged with other Kconfig models with the source keyword. This allows a Kconfig model to be assembled from smaller models.

A feature can have one of the following types: bool (Boolean), string, int (integer), hex (hexadecimal), and tristate. The tristate type allows source code to be compiled as modules and loaded at run-time instead of statically linked to the program binary [41].

Fig. 2 is a simple example. A feature is defined with the keyword 'config' along with its type, constraints, and subfeatures as its nested attributes. BOOL1, BOOL2, LITERAL, INVISIBLE, ALT1, and ALT2 are the features of this model. BOOL1 and LITERAL have their default value defined with the keyword 'default'.

Constraints are feature dependencies. The following constraints are supported in Kconfig:

- **prompt** denotes whether a feature can be selected by the user, shown as its description in double quotes after the type declaration or after the keyword 'prompt'.
- **depends on** means a feature can be selected only if its condition is satisfied. A *condition* is one or more bool or tristate features, with operators and (&&), or (||), equals (==), unequals (!=), and not (!).
- **select** means the selection of a feature forces the selection of another bool or tristate feature. During configuration, select can override prior decisions made to the feature being selected. Note: forcing the deselection of a feature is not possible.
- **choice** means only one of the nested bool or tristate features can be selected. choice itself can be optional, which allows no feature to be selected.
- **if** guards a statement or a block of statements. If the guard is not satisfied, the statement or block is ignored.
- **menu** defines hierarchies among features, so that selecting a menu feature enables the selection of its nested features.

A more detailed description of Kconfig is given in [2], [41].

In Fig. 2, INVISIBLE cannot be selected by the user as it does not have a prompt, but it *can* be selected when (1) BOOL1 is selected OR (2) BOOL2 is selected and BOOL1 is *not* selected. LITERAL is a string feature which the user can set its value only when BOOL1 is selected. ALT1 and ALT2 are alternative features, where one can be selected only when BOOL2 is selected.

Although the above concepts seem well-defined, Kconfig has ambiguities, which we describe in Sec. III.

[2]defconfig selects features by default values. randconfig chooses features randomly. allyesconfig chooses as many features as possible.

```
1  menu "Parent feature"
2      config BOOL1
3          bool "Boolean feature 1"
4          select INVISIBLE
5          default y
6
7      config BOOL2
8          bool "Boolean feature 2"
9          select INVISIBLE if !BOOL1
10
11     config LITERAL
12         string "String feature"
13         depends on BOOL1
14         default "string value"
15
16     config INVISIBLE
17         bool
18 endmenu
19
20 if BOOL2
21     choice
22         bool "Alternative features"
23
24         config ALT1
25             bool "Alternative 1"
26
27         config ALT2
28             bool "Alternative 2"
29     endchoice
30 endif
31 source "./anotherKconfigmodel.in"
```

Fig. 2: Example of a Kconfig model.

E. Converting a Kconfig Model into a Propositional Formula

As with feature models, a Kconfig model can be converted into a propositional formula. Few existing tools are capable of extracting the precise semantics of Kconfig models and converting them into a format usable for automated solvers [6], [9], [42]. The process to convert a Kconfig model into a propositional formula is similar for all tools:

1) Modify or extend the parser within Kconfig to obtain the features and their constraints in a simplified format;
2) Encode logical information into an intermediate model suitable for analysis;
3) Transform Kconfig constraints of the intermediate model into a set of propositional clauses. The conjunction of these clauses becomes the propositional formula ϕ for the Kconfig model; and
4) Convert ϕ into ϕ_{es}^{cnf} and encode ϕ_{es}^{cnf} as a dimacs file, a widely used format for automated solvers [43].

Depending on the purpose of the tool, features and constraints may be omitted or added.

Among these tools, kconfigreader is known to capture more Kconfig semantics than others [44]. Unfortunately, kconfigreader uses the ET from Plaisted & Greenbaum [38], whose consequences we explore later.

III. KCLAUSE : KCONFIG MODEL TO CNF FORMULA

Kclause is a new tool to convert a Kconfig model into a CNF formula. To create a compact and equivalent formula, Kclause:

1) Parses Kconfig model(s) to extract the list of features and their constraints;
2) For each feature F, converts the constraints restricting F into equivalent CNF clauses; and
3) Conjoins the CNF clauses to produce ϕ^{cnf}, as all clauses must be satisfied for a valid configuration.

We describe these steps in more detail below.

A. Converting Constraints into Propositional Clauses

A basic approach to derive ϕ converts each constraint into propositional clauses and conjoins them. Consider a Kconfig model with two constraints:

$$\texttt{S select A, B depends on D} \tag{3}$$

select *restricts* A and depends on *restricts* B, meaning that the value of A is determined by the value of S and the value B is determined by the value of D. Here the select constraint means $(S \Rightarrow A)$ and the depends on constraint means $(B \Rightarrow D)$. As both constraints must be satisfied, the resulting formula is a conjunction of the two propositional clauses: $(S \Rightarrow A) \wedge (B \Rightarrow D)$. Simple.

Unfortunately, the semantics of other Kconfig constraints are less obvious. Consider a slight change to (3) where feature A is the target of both restrictions:

$$\texttt{S select A, A depends on D} \tag{4}$$

The above does ***not*** mean $(S \Rightarrow A) \wedge (A \Rightarrow D)$. We deciphered (4) by using small Kconfig models to deduce a truth table to reveal its semantics. We discovered (4) to mean:

$$s4 \;=\; (S \Rightarrow A) \wedge (\neg S \Rightarrow (A \Rightarrow D)) \tag{5}$$

which is *decidedly* not evident from the Kconfig manual [41].

Continuing, a variation of (4) is:

$$\texttt{S select A if C, A depends on D} \tag{6}$$

whose meaning is: $(C \Rightarrow s4) \wedge (\neg C \Rightarrow (A \Rightarrow D))$, where s4 comes from Eq. (5). Again this translation is not obvious.

Kconfig does seem to follow general rules. When multiple constraints restrict the same feature, they may need to be considered semantically together for conversion. We performed an analysis of Kconfig to identify such combinations. Our analysis is not exhaustive; we identified combinations that were used in the Kconfig models that we analyzed in this paper.[3] We continue to generalize our analysis as we incrementally develop our tool. When a combination of constraints is encountered that we have not considered (see "indexed" in the next section), Kclause raises a warning. In Sec. V we present experimental results that show we accurately capture the semantics of our Kconfig models with high confidence.

The problem, of course, is that the Kconfig language is ambiguous. The behavior of some constraint combinations are not defined in the Kconfig manual, so their use can result an invalid model or be interpreted differently by system developers [44]. Some combinations are actually recommended ***not*** to be used but are used by system developers anyways; Kconfig raises no warnings or errors when such combinations are used [41].

Frankly, we are surprised that Kconfig is used given the above observations. A need for a well-defined semantics was raised in Kconfig documentation [2].

[3]Different values of string, int, and hex features were not considered, as representing them in propositional formulas were beyond the scope of our paper. Instead, we assumed their default values were used when selected.

B. Deriving a CNF formula

Converting a propositional formula into an equivalent CNF formula can increase the number of clauses exponentially [36]. Prior work used ETs to avoid this. We exploited the structure of Kconfig constraints, rather than ϕ, to create a compact and equivalent ϕ^{cnf} instead.

Let κ_f be one or a combination of constraints restricting a feature f and let ρ_f be the conjunction of propositional clauses that represent κ_f. As ϕ is a conjunction of ρ_f for each feature, we can convert each ρ_f into equivalent CNF clauses, ρ_f^{cnf}, and conjoin them for each feature to get ϕ^{cnf}.

As we have defined finite legal combinations of constraints for κ_f, legal clauses for ρ_f can be predefined. We precomputed ρ^{cnf} for possible combinations of constraints so that it can be retrieved from a table "indexed" by κ_f. Doing so, we do not have to use a CNF transformation tool when converting an actual Kconfig model. For any given κ_f from a Kconfig model, Kclause looks up its precomputed ρ^{cnf} and substitutes the variables with actual feature names to derive ρ_f^{cnf}.

Then, to make ϕ^{cnf} compact, we simplified ρ^{cnf} to have as few clauses as possible. Unlike arbitrary formulas, simplifying ρ^{cnf} is feasible as it has predefined structure and limited number of clauses.

To illustrate, recall the example on the previous section:

$$\kappa = \{\texttt{S select A if C, A depends on D}\}$$

Its propositional formula derived as:

$$\rho = (C \Rightarrow ((S \Rightarrow A) \wedge (\neg S \Rightarrow (A \Rightarrow D)))) \wedge (\neg C \Rightarrow (A \Rightarrow D))$$

Its simplified ρ^{cnf} is $(\neg C \vee \neg S \vee A) \wedge (S \vee \neg A \vee D) \wedge (C \vee \neg A \vee D)$, consisting of 4 variables and 3 clauses. Formulas of this size are easy to simplify by the WolframAlpha Boolean algebra simplifier [45].

In contrast, Tseitin's transformation of ρ results in 10 variables and 20 clauses as:

$$\begin{gathered}\rho_{es}^{cnf} = (\neg x_1 \vee \neg A \vee D) \wedge (x_1 \vee A) \wedge (x_1 \vee \neg D) \\ \wedge (\neg x_2 \vee \neg S \vee A) \wedge (x_2 \vee S) \wedge (x_2 \vee \neg A) \\ \wedge (\neg x_3 \vee S \vee x_1) \wedge (x_3 \vee \neg S) \wedge (x_3 \vee \neg x_1) \\ \wedge (\neg x_4 \vee C \vee x_1) \wedge (x_4 \vee \neg C) \wedge (x_4 \vee \neg x_1) \\ \wedge (\neg x_5 \vee x_2) \wedge (\neg x_5 \vee x_3) \wedge (x_5 \vee \neg x_2 \vee \neg x_3) \\ \wedge (\neg x_6 \vee \neg C \vee x_5) \wedge (x_6 \vee C) \wedge (x_6 \vee \neg x_5) \\ (x_4) \wedge (x_6)\end{gathered}$$

Although no redundant solutions are created, Tseitin's result uses many more variables and clauses. Again, the value of our approach is that it produces more compact formulas than prior work, which in turns reduces sampling times. This improvement is documented in Sec. V.

IV. SMARCH : SAMPLING SPL SPACES

A. URS of a Configuration Space using #SAT

Instead of using a CBBD, we devised a sampling algorithm that uses #SAT, which counts the number of solutions for a propositional formula and is scalable enough to analyze Kconfig models. #SAT solvers extend SAT algorithms by associating the number of solutions with each truth assignment [33]. We use sharpSAT [5], a state-of-the-art #SAT solver.

Creating a one-to-one mapping between integers and configurations is accomplished by recursively partitioning $\mathcal{C}(\phi)$ on variable assignments. A variable $v \in \phi$ partitions $\mathcal{C}(\phi)$ into disjoint spaces $\mathcal{C}(\phi \wedge \neg v)$ and $\mathcal{C}(\phi \wedge v)$. sharpSAT can compute $|\phi \wedge \neg v|$ and $|\phi \wedge v|$.

Given a randomly-selected integer $r \in [1..|\phi|]$, if $r \leq |\phi \wedge \neg v|$ the $\mathcal{C}(\phi \wedge \neg v)$ space is selected for recursive partitioning, otherwise $\mathcal{C}(\phi \wedge v)$ is selected and $|\phi \wedge \neg v|$ is subtracted from r to adjust the search in $\mathcal{C}(\phi \wedge v)$. This process is repeated using other variables in ϕ, until all variables are assigned a value.

Doing so creates a tree structure where a branch denotes a variable assignment and each node stores the size of a partition. Sampling involves traversing the tree from the root to a leaf based on r, reusing portions of previous traversals as much as possible. This approach has following properties:

- The returned configuration is valid, as invalid variable assignments result in an empty partition which cannot be selected;
- The mapping is one-to-one as assigning truth values for all variables results a partition of size one, while different r cannot return the same configuration as they will select different partitions; and
- The mapping is unbiased towards certain variable selections, as selecting a partition to recurse is determined by r and the size of partitions.

The partitioned formula can become empty after *Boolean Constraint Propagation* (BCP) by the assigned variables, where the size of the partition is 2^u where u=# of unassigned variables. In this case, unassigned variables are *free variables* — variables without constraints. Assigning any value to a free variable does not affect the validity of a solution. When this situation arises, values of free variables are assigned to binary values of r without further partitioning.

B. Enhancing efficiency of sampling with cube-and-conquer

The run-time of the above algorithm is affected by:

- The number of recursions, as each recursion involves at least one sharpSAT call which is expensive and
- The complexity of the formula for each partition, as a more complex formula takes more time for sharpSAT.

Some variables are more constraining than others, so using them to partition first reduces the complexity of the formula more than other variables. Smarch uses these variables first to reduce the number of recursions and sharpSAT solving times.

We use the partitioning of the *cube-and-conquer* ($\mathbb{CC}$) algorithm of Heule *et al.* [46]. $\mathbb{CC}$ was originally devised to increase the scalability of SAT solving. It finds variables in ϕ that can eliminate the number of clauses the most by BCP.

Further, n variables create 2^n partitions, where each partition can be solved by a SAT solver. $\mathbb{CC}$ reduces the number of clauses for each partition as much as possible, while preventing free variables from being used for partitioning.

As $\mathbb{CC}$ reduces the number of clauses, #SAT run times are reduced. Further, since free variables do not affect the complexity of the formula, they are deferred to the end of the recursion and assigned all at once. This also reduces the number of recursions.

Alg. 1 is the Smarch algorithm. First, the algorithm counts total number of solutions for ϕ and generates random numbers (lines 2, 3). Each random number is used to sample a solution by recursive variable assignments (lines 4–7). At each recursion, the algorithm checks if all unassigned variables are free variables (line 11). If so, their values are assigned without further recursion (lines 12–17). If not all variables are free and there are remaining unassigned variables (line 18), $\mathbb{CC}$ finds variables that can be used to partition the formula (line 19). We find a maximum of five variables from $\mathbb{CC}$, as more variables increase the number of partitions exponentially and reduces efficiency.[4] This creates a maximum of 32 partitions, where one partition is selected to recurse based on r (lines 20–26). The recursion continues with the selected cube (lines 27).

A key property of Smarch that we have not exploited yet is that it can be executed in parallel. When running sharpSAT for large ϕ, $\mathbb{CC}$ partitions the formula so that the partitions can be counted in parallel. The total number of solutions is the sum of the counts from each partition. Further, sampling multiple configurations can be parallelized as soon as random numbers are generated. A subset of random numbers can be sampled with one machine, while the others can be sampled with another machine without sharing data.

V. EVALUATION

Our paper covers multiple topics each requiring its own set of research questions. The first topic is the quality of propositional formulas produced by Kclause; there are other tools that can perform the same task. How does Kclause compare? The second is the quality of "random" selections of solutions to a propositional formula. Smarch performs URS; there are other tools that can perform the same task. How does Smarch compare? And finally, we present a demonstration of Kclause+Smarch in action.

Research Questions to Evaluate Kclause. We compare Kclause to kconfigreader (kcr), the best tool-to-date for converting a Kconfig model to a propositional formula. Other tools have been built [6], [9], [42], but kcr is considered to be the most accurate and currently maintained [44]. We ask:

RQ1: Which tools make formulas equivalent to Kconfig models?

[4]Finding an optimal number of variables is future work.

Algorithm 1: `Smarch` Algorithm

```
1  Procedure SMARCH(n, φ):
       Input  : n (number of samples)
                φ (propositional formula)
       Output: sSet (set of n samples)
2      total ← number of solutions for φ;
3      rSet ← n distinct random integers from [1, total];
4      for each r in rSet do
5          s ← new empty set;
6          s ← SAMPLE(r, φ, s);
7          add s to sSet;
8      return sSet;
9
10 Procedure SAMPLE(r, φ, s):
       Input  : r (a random integer)
                φ (propositional formula)
       Output: s (sample = set of variable assignments)
11     if BCP(φ, s) is empty then
12         for all variables unassigned in BCP(φ, s) do
13             i ← r%2; r ← r/2;
14             if (i == 0) then
15                 add ¬v to s;
16             else
17                 add v to s;
18     else if unassigned variables remain then
19         cubes ← variable assignment sets from CC(φ ∧ s);
20         for each cube in cubes do
21             cs ← number of solutions for φ ∧ s ∧ cube;
22             if (r ≤ cs) then
23                 add cube to s;
24                 break;
25             else
26                 r ← r − cs;
27         s ← SAMPLE(r, φ, s);
28     return s;
```

RQ2: Do redundant configurations affect `URS`?
RQ3: Do fewer variables and clauses reduce `URS` time?

Research Questions to Evaluate `Smarch`. We compare `Smarch` to `Unigen2` [30] and `randconfig`, tools that also randomly sample configurations from a `Kconfig` model or a propositional formula. We ask:

RQ4: Which tools can sample `Kconfig` models?
RQ5: Which tools generate uniformly distributed samples?

Research Question to Evaluate `Kclause`+`Smarch`. Finally, we ask how `Kclause` and `Smarch` can be used together. We ask:

RQ6: Can `Kclause+Smarch` analyze colossal spaces?

Systems used in our Experiments. Table II lists the `Kconfig` systems used by others and that we reuse in our experiments. We used the latest version of each system. We italicize their *names* to differentiate them from tools that we evaluate.

Name	Ref	Description	Version
axTLS	[47]	Client/server library for embedded systems	2.1.4
Toybox	[48]	Linux command line utilities for Android	0.7.5
Fiasco	[49]	Real-time microkernel	17.10
Busybox	[50]	Single executable for UNIX common utilities	1.28.0
uClibc-ng	[51]	Lightweight C library for embedded Linux systems	0.9.33.2

TABLE II: `Kconfig` Systems Used in Our Evaluation

Our experiments rely on this fact: `Smarch` uses a one-to-one correspondence between integers in $[1..|\phi|]$ and configurations in $\mathcal{C}(\phi)$. A `URS` of integers in $[1..|\phi|]$ yields a `URS` of configurations in $\mathcal{C}(\phi)$, and vice versa. That is, by construction `Smarch` performs `URS`.

We used Intel `i7-6700@3.4Ghz` Linux machine with `16GB` of RAM for all experiments.

A. Evaluation of `Kclause`

5.1.1 RQ1: Which tools make formulas equivalent to `Kconfig` models?

Let τ be either `Kclause` or `kcr`. A τ-produced formula ϕ_τ is *equivalent* to a `Kconfig` model μ if Eq. (1) holds, namely $\mathcal{C}(\mu) \equiv \mathcal{C}(\phi_\tau)$. As `Kclause` was created to determine `Kconfig` semantics, a formal proof of equivalence is not possible and enumerating all solutions for any of our `Kconfig` models is infeasible. Instead, we conducted the following experiment:

T1 Check whether ϕ_τ-sampled solutions are also μ solutions. `Smarch` randomly samples $\mathcal{C}(\phi_\tau)$. Each sample is validated using `Kconfig` as a test oracle. A sample *passes* if `Kconfig` raises no errors and makes no corrections.

T2 Check whether `Kconfig`-sampled μ solutions are also unique ϕ_τ solutions. We used `Kconfig`'s built-in tool, `randconfig`, to generate "random" samples from $\mathcal{C}(\mu)$. A test formula ρ for each sample is the conjunction of ϕ_τ and the variable assignments of the sample. Using `sharpSAT`, a `randconfig` sample:

- *passes* if $|\rho|=1$,
- is *redundantly represented* if $|\rho|>1$, or
- *fails* if $|\rho|=0$.

We collected these measurements:

- **T1pass** is the % of samples that passed **T1**.
- **T2pass** is the % of samples that passed **T2**.
- **T2fail** is the % of samples that failed **T2**.
- $|\phi_\tau|$ is counted by `sharpSAT`.

Each sample is a Bernoulli (coin flip) experiment. We evaluated **T1** and **T2** each with `1068` samples to see if `Kclause` and `kcr` pass **T1** and **T2** with `95%` confidence and `3%` margin of error [52].

`Kclause` results are in Table III. `Kclause` formulas passed **T1** and **T2** for all systems.

	T1pass	T2pass	T2fail	$\lvert\phi_{\texttt{Kclause}}\rvert$
axTLS	100%	100%	0%	2.0×10^{12}
Toybox	100%	100%	0%	1.4×10^{81}
Fiasco	100%	100%	0%	1.0×10^{10}
Busybox	100%	100%	0%	1.3×10^{248}
uClibc-ng	100%	100%	0%	8.0×10^{36}

TABLE III: `Kclause` vs. `Kconfig`

Next, we evaluated `kcr`, with results in Table IV, and observed:[5]

[5]Since `kcr` formulas introduce features to encode different values of non-boolean features, we added constraints so that only the default value of non-boolean features are selectable for a fair comparison.

- **T1pass** says that configurations in $\mathcal{C}(\phi_{\texttt{kcr}})$ are valid configurations of $\mathcal{C}(\mu)$ with `95%` confidence and `3%` margin of error.
- **T2pass** says formulas from `kcr` produce redundant configurations due to ETs. Only *Toybox* passed **T2**, as `kcr`'s translation introduced no additional variables.
- **T2fail** for *Fiasco* reveals corner cases of `Kconfig` models that `kcr` does not translate correctly.

	T1pass	T2pass	T2fail	$\lvert\phi_{\texttt{kcr}}\rvert$
axTLS	100%	82.6%	0.0%	1.2×10^{13}
Toybox	100%	100.0%	0.0%	1.4×10^{81}
Fiasco	100%	65.4%	1.3%	1.8×10^{10}
Busybox	100%	16.1%	0.0%	1.5×10^{249}
uClibc-ng	100%	0.0%	0.0%	4.7×10^{38}

TABLE IV: `kcr` vs. `Kconfig`

Conclusion: ***`Kclause` produces equivalent formulas to `Kconfig` models with `95%` confidence and `3%` margin of error. `kcr` does not derive equivalent formulas to `Kconfig` models.***

5.1.2 RQ2: Do redundant configurations affect `URS`?

Let $\mathcal{S}_{\texttt{kcr}}$ be a set of `n` configurations sampled by `URS` from $\mathcal{C}(\phi_{\texttt{kcr}})$. If redundant configurations do not affect uniformity, $\mathcal{S}_{\texttt{kcr}}$ should be uniformly distributed in $\mathcal{C}(\phi_{\texttt{Kclause}})$.

We used the *Kolmogorov–Smirnov* (`KS`) [53] test on order statistics [54] to answer this question. A `KS` test checks whether two data sets are sampled from the same distribution [53]; the closeness of the match is a parameter of the test. We used `95%` confidence in all `KS` tests — meaning that a `KS` test returns affirmative when two samples from the same distribution is higher than `95%`.

Here is how we applied a `KS` test: Suppose $\mathcal{S}_{\texttt{kcr}}$ are uniformly distributed over $\mathcal{C}(\phi_{\texttt{Kclause}})$. We use the term $\texttt{k}^{\texttt{th}}$*-best sample* as being the sample with the $\texttt{k}^{\texttt{th}}$-best performance in `n` samples. For the $\texttt{k}^{\texttt{th}}$-best sample in $\mathcal{S}_{\texttt{kcr}}$, let $\texttt{r}_\texttt{k}$ be its rank in a fixed-ordering of $\mathcal{C}(\phi_{\texttt{Kclause}})$. Order statistics state that the expected rank of $\texttt{r}_\texttt{k}$ is $\frac{\texttt{k}}{\texttt{n}+1}$, so that the `n` samples of $\mathcal{S}_{\texttt{kcr}}$ should have equally-spaced ranks [54]. The uniformity of $\mathcal{S}_{\texttt{kcr}}$ samples is determined by how close each rank $\texttt{r}_\texttt{k}$ matches its expected rank $\frac{\texttt{k}}{\texttt{n}+1}$ for $1\leq\texttt{k}\leq\texttt{n}$.

Deriving $\texttt{r}_\texttt{k}$ by enumerating all configurations is infeasible due to colossal size of `Kconfig` spaces. Instead, we utilized the binary tree sampling structure of `Smarch`. Given a formula ϕ, `Smarch` takes a sampled number $1\leq\texttt{r}_\texttt{k}\leq|\phi|$ and finds its matching configuration $\texttt{c}_\texttt{k}\in\mathcal{C}(\phi)$ by traversing `Smarch`'s tree using $\texttt{r}_\texttt{k}$. Ranking — which is what we need here — is the inverse: `Smarch` is given $\texttt{c}_\texttt{k}$ and finds its $\texttt{r}_\texttt{k}$ by traversing the tree based on $\texttt{c}_\texttt{k}$'s variable assignments.

For each system, we sampled `100`, `500`, and `1000` configurations independently from a `kcr` formula using `Smarch`. For each sample set, we derived the ranks and performed a `KS` test. If a `KS` test passes, we can say $\mathcal{S}_{\texttt{kcr}}$ is uniformly distributed in $\mathcal{C}(\phi_{\texttt{Kclause}})$ with `95%` confidence.

Our results are the two-dimensional array of graphs of Fig. 3. Each row is the same system at different sample sizes; each column shows different systems with the same sample size. Each graph plots the actual and theoretical rank of the samples sorted by their actual rank, along with the `KS` test result.

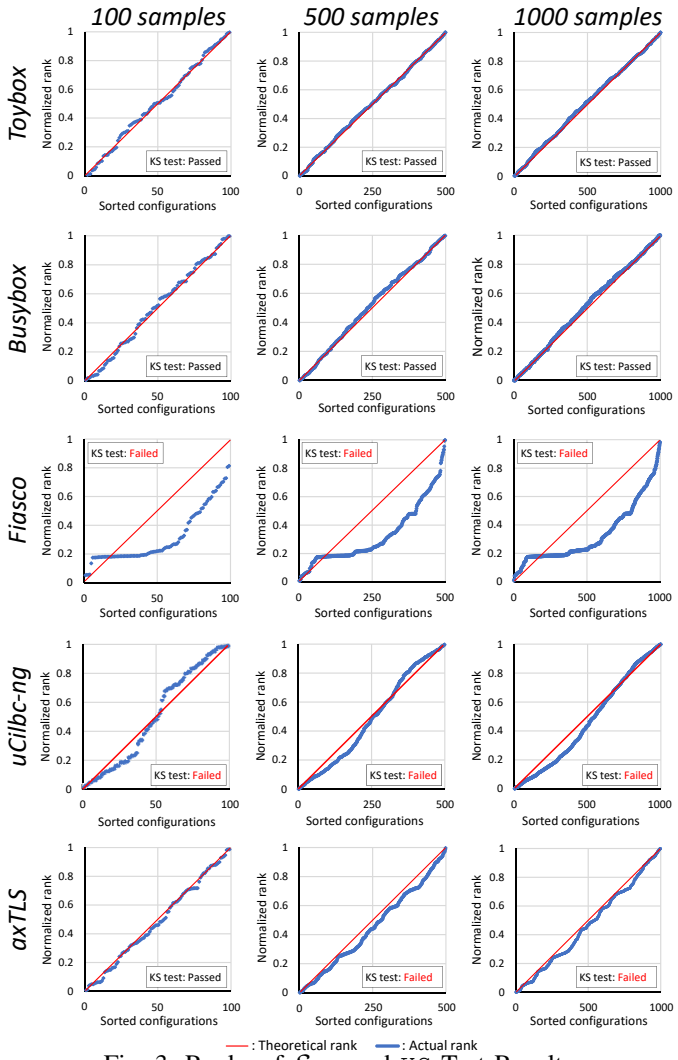

Fig. 3: Ranks of $\mathcal{S}_{\texttt{kcr}}$ and `KS` Test Results.

Our observations on `kcr`:

- The shape of the graphs for each system (from 100 to 1000 samples) are virtually identical; the only difference is the increasing fidelity of the true dispersion of `kcr` samples with increasing sample set size.
- *Toybox* and *Busybox* passed every `KS` test,
- *Fiasco* and *uClibc-ng* failed every `KS` test, and
- *axTLS* passed the `KS` test with a sample set of size `100`. As more samples revealed the true distribution [54], we found `kcr` samples of *axTLS* configurations are *not* uniformly distributed.

In short: A `KS` test passes if:

- there are no redundant configurations (*eg., Toybox*) or
- redundant configurations are distributed evenly enough not to be noticed (*eg., Busybox* at a `1000` sample granularity).

Of course, the problem is one does not know when either of

these conditions hold — except for Kclause which guarantees no redundant configurations.

Conclusion: Propositional formulas should not represent configurations redundantly as URS will likely be biased.

5.1.3 RQ3: Do fewer variables and clauses reduce URS time?

We compared formulas produced by Kclause and kcr to see which is more compact and more efficient for sampling. We used following measurements:

- |V| is the number of variables in the formula. Note kcr omits free variables from dimacs files. We included them in |V| for a fairer comparison.
- |C| is the number of clauses in the formula.
- T is the time taken to sample a configuration using Smarch, in seconds. T is averaged over 1068 samples.

Table V shows the results. The %impr columns show the percentage improvement in performance delivered by Kclause over kcr for a given column metric, computed as $\frac{kcr_{column} - Kclause_{column}}{kcr_{column}}$.

System	Kclause			kcr			%impr		
	|V|	|C|	T	|V|	|C|	T	|V|	|C|	T
axTLS	94	190	1.28	191	732	3.73	51%	74%	66%
Toybox	316	106	1.53	324	212	1.77	2%	50%	14%
Fiasco	234	1178	1.01	571	3810	2.57	59%	69%	61%
Busybox	998	962	19.04	1158	2287	27.75	14%	58%	31%
uClibc-ng	269	1403	3.95	576	3826	7.65	53%	63%	48%

TABLE V: Formula Compactness Comparison

We observe *on average*:

- Kclause used 36% fewer variables than kcr,
- Kclause used 63% fewer clauses than kcr, and
- Kclause completed a sampling 44% faster than kcr.

Conclusion: Kclause generates formulas with fewer variables and clauses, and with 44% faster sample times, on average, than kcr.

B. Evaluation of Smarch

Kconfig models have colossal product spaces. We ask two questions in this section: which tools can sample such spaces? And which can URS these spaces?

5.2.1 RQ4: Which tools can sample Kconfig models?

Smarch samples solutions of a given propositional formula. Other tools can also do this and use very different sampling technologies:

- Unigen2 [30] partitions a solution space using an approximate #SAT solver [31]. When partitions are small enough, solutions in a partition are enumerated for random selection.
- randconfig [2] randomly assigns truth values to features. To avoid invalid configurations, features assigned a value may restrict values of features not yet assigned.

We measured the time in seconds spent to sample a configuration for a Kconfig system on a single core. Time was averaged over 1068 samples for a 95% confidence and 3% margin of error. If any sample took more than 30 minutes, we considered it a *time-out*. Table VI shows the average time for different systems and tools, in seconds.

	Smarch	Unigen2	randconfig
axTLS	1.28	277.39	1.00
Toybox	1.53	time-out	1.03
Fiasco	1.01	0.21	1.35
Busybox	19.04	time-out	2.12
uClibc-ng	3.95	time-out	1.31

TABLE VI: Sampling Time Comparison

Observations:

- Smarch completed all samples in reasonable time.
- Unigen2 was very fast for *Fiasco*, very slow for *axTLS* and timed-out for the remaining systems.[6]
- randconfig was generally the fastest as it only needs to read through the Kconfig models to sample a configuration. Note that, randconfig requires at least 1 second delay to avoid sampling redundant configurations.

We were surprised at the failure of Unigen2, as it could sample configurations from a formula with more than 100,000 variables in under a minute [30]. Looking further, we discovered something unexpected: some systems that Unigen2 sampled in [30] Smarch could *NOT* sample, and vice versa. That is, Smarch could sample the colossal spaces of all of our SPLs, but Unigen2 could not. Unigen2 was limited to *axTLS* and *Fiasco* that had rather small product spaces $O(10^{13})$.

The reason for Unigen2's failure is unclear. We suspect that Unigen2 may perform well with highly constrained formulas, where variables have more clauses constraining their values. For the formulas used in the Unigen2 paper [30] that sharpSAT could count, their average $|\phi|$ was $O(10^{13})$, *axTLS*- and *Fiasco*-sized. As Unigen2 enumerates solutions of a partition to randomly sample a solution among them, we suspect that their enumeration may be costly for formulas with colossal $|\phi|$.

We looked further: *Fiasco* was the only system that Unigen2 showed its expected performance, which had the smallest $|\phi|$. We discovered that increasing $|\phi|$ by arbitrary removing half of the clauses from *Fiasco*'s formula or by arbitrary adding free variables makes Unigen2 time-out as well.[6]

Conclusion: randconfig and Smarch can sample colossal Kconfig product spaces; Unigen2 cannot.

5.2.2 RQ5: Which tools generate uniformly distributed samples?

We know Smarch performs URS by construction. We use Smarch to evaluate whether or not Unigen2 and randconfig can URS as well.

Let τ be Unigen2 or randconfig. Let $\mathcal{S}_\tau$ be a set of n configurations sampled from $\mathcal{C}(\phi_{Kclause})$ by τ. We use a KS test, using Smarch to compute the ranks of $\mathcal{S}_\tau$ and then

[6]We have confirmed our results with the developer of Unigen2.

compare these ranks to the expected ranks of order statistics, just as in **RQ2**.

For each experiment, we sampled set of `100`, `500`, and `1000` configurations independently from a `Kclause` formula using τ. For each sample set, we derived the rank and performed a `95%` confidence `KS` test.

Let τ be `Unigen2`. We performed experiments on *Fiasco*, which was the only system that `Unigen2` could sample in a reasonable time, and one additional system that `Smarch` could evaluate from [30].

Fig. 4 shows our results. `Unigen2` passed all `KS` tests for all systems and sample set sizes, indicating that `Unigen2` samples are indeed uniformly distributed. ***For these systems, we confirm that `Unigen2` performs `URS` on the propositional formulas that it is given with `95%` confidence.***

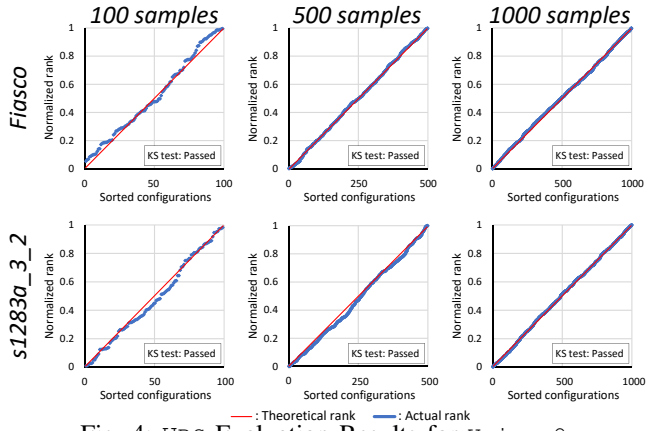

Fig. 4: `URS` Evaluation Results for `Unigen2`.

Now let τ be `randconfig`, a tool of `Kconfig`. Technically `randconfig` does not sample $\mathcal{C}(\phi_{\texttt{Kclause}})$ but its equivalent $\mathcal{C}(\mu)$, where μ is a `Kconfig` model.

Fig. 5 shows our results for all five systems. `randconfig` was unable to pass any `KS` test for these systems. indicating that `randconfig` is biased toward selecting certain features. ***We conclude that `randconfig` cannot be used when `URS` is needed.***

5.2.3 Summary

Only `Smarch` could `URS` colossal spaces. `randconfig` sampling is not uniform. `Unigen2` did not scale to spaces exceeding `O`(10^{13}).

C. Evaluation of `Kclause+Smarch`

5.3.1 RQ6: Can `Kclause+Smarch` analyze colossal spaces?

We now show `Kclause+Smarch` can be used to analyze `Kconfig` systems, which was not possible before.

Oh *et al.* [26] analyzed PCS graphs to explore a configuration space and find near-optimal configurations for given performance metrics. A PCS graph plots the performance of sampled configurations from best performance to worst. PCS graphs are a useful way to visualize the shape of a configuration space and how a feature or combination of features influence performance [26], [55].

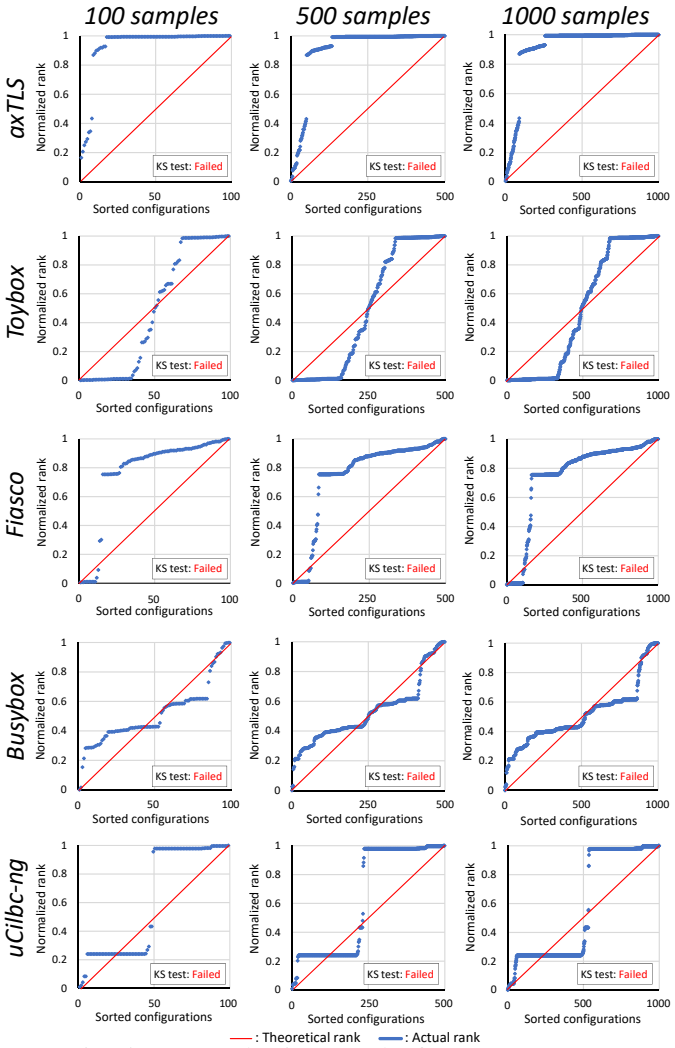

Fig. 5: `URS` Evaluation Results for `randconfig`.

Near-optimal configurations could be found by taking `n URS`. The sample with the best performance has average rank of $\frac{1}{n+1}$ within the configuration space, independent of the size of the space and metric [26]. So the best performing configuration in `99 URS` will, on average, be in the top `1` percentile of *all* configurations.

The approach of Oh *et al.* could not analyze `Kconfig` systems as `CBBD`s do not scale. We used `Smarch` and `randconfig` instead of their sampling algorithm, to plot the PCS graphs of `Kconfig` systems.

We took `1000` samples each using available tools, `Kclause+Smarch` and `randconfig`. Then we built each sample and measured its build size, a metric non-trivial to estimate and less vulnerable to measurement noises.[7] Then, we sorted the samples according to their build size and plotted them as a PCS graph.

Fig. 6 shows the PCS graphs of each `Kconfig` system from `Smarch` samples (**blue line**) and `randconfig` samples

[7]We included or excluded some features from sampling to ensure builds without errors. This is not a problem with `Kconfig` but rather `Kconfig` systems where some configurations have build errors, leading to invalid build sizes. The list of features are in the data archive [56].

(**orange line**).

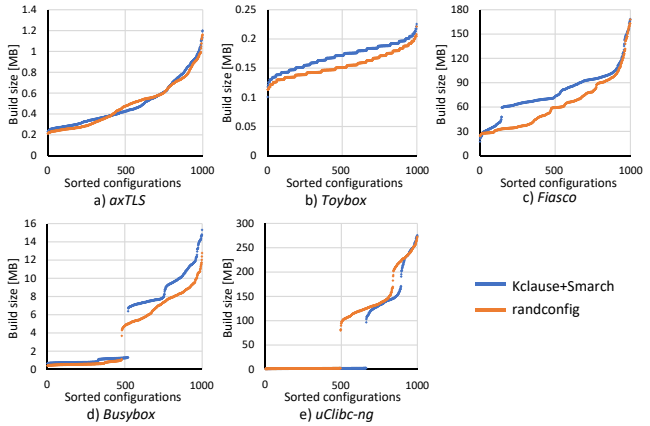

Fig. 6: PCS Graphs of `Kconfig` system Build-Sizes.

So which PCS graph is correct? To answer this, we performed the following experiment. Let S_{sm} and S_{rc} be identically-sized sets of samples from `Smarch` and `randconfig`, respectively. For a feature `f`, we measured:

- r_f = $|\phi \wedge f|$ / $|\phi|$, the true fraction of configurations with `f`, computed by `sharpSAT`,
- r_f^{sm} = (# of samples with `f` in S_{sm}) / (# of samples in S_{sm}), as the estimated ratio from `Smarch` samples, and
- r_f^{rc} = (# of samples with `f` in S_{rc}) / (# of samples in S_{rc}), as the estimated ratio from `randconfig` samples.

From these measurements, we derived:

- $e_f^{sm} = |r_f - r_f^{sm}|$, as the estimation error for `Smarch`, and
- $e_f^{rc} = |r_f - r_f^{rc}|$, as the estimation error for `randconfig`.

We measured e_f^{sm} and e_f^{rc} for all features in each `Kconfig` system and derived their average ($avg(e_f^{sm})$, $avg(e_f^{rc})$) and maximum ($max(e_f^{sm})$, $max(e_f^{rc})$). We also used different sample sizes: 100, 500, and 1000. Fig. 7 shows the results, where the X-axis is the number of samples and the Y-axis is the estimation error.

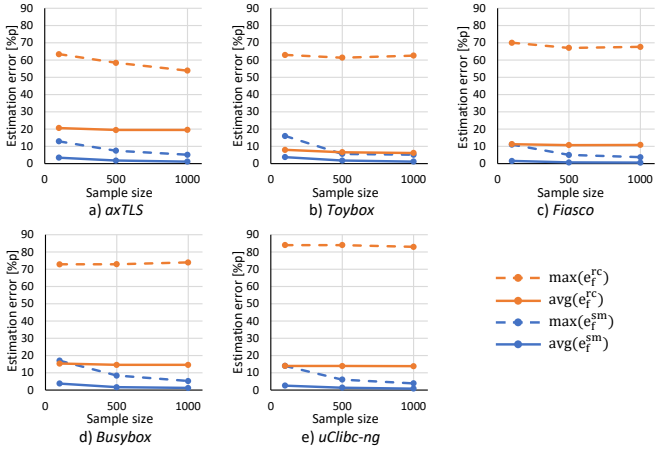

Fig. 7: Estimation Error of `Smarch` and `randconfig` Samples.

We observed:

- `Smarch` errors were much lower than `randconfig` on all metrics.
- `Smarch` errors decreased as sample size increased as expected, whereas `randconfig` did not show this trend.
- `randconfig` was not able to accurately estimate r_f. S_{rc} often had no samples with certain features, while r_f implies it should.

We conclude that `Kclause`+`Smarch` yields accurate statistics on `Kconfig` systems.

D. Threats to Validity

Internal Validity. For **RQ1**, **RQ3**, and **RQ4**, we conducted tests 1068 times to control randomness and derive statistical bounds on averaged results. For **RQ2** and **RQ5**, we used different sample sizes to control the randomness. For **RQ6**, we verified that all selected configurations could be built. We measured their build size as it is less susceptible to noises.

External Validity. We used 5 real-world systems with different domains, numbers of features, and number of clauses. These systems used different combinations of constraints and exercised different parts of our tools. We are aware that the accuracy of the `CNF` formula and scalability of `URS` may not generalize to all systems using `Kconfig`, but identical trends across systems from our evaluations gives confidence that our conclusions should hold for many `SPL`s.

VI. Related Work

A. Extracting Semantics of `KConfig` Models

Prior work also tried to capture the semantics of `Kconfig` models for different analysis purposes. Berger *et al.* [9] developed `LVAT` to analyze the structure of `Kconfig` models. Dietrich *et al.* [6] developed `Undertaker` to find dead code blocks — blocks that no configuration uses. Kästner *et al.* [42] developed `kcr` for the TypeChef tool [57], which checks the consistency between a feature model and `SPL` source code. El-Sharkawy *et al.* [44] analyzed corner cases of `Kconfig` semantics and evaluated the above tools for correctness, where `kcr` was the most accurate among them.

`Kclause` is different as it is aimed at `URS`, which requires more demanding properties of propositional formulas. Prior work mostly used small test cases to evaluate their accuracy, while we used real systems using `Kconfig`.

B. Automated Reasoning on `Kconfig` Systems

`Kconfig` models have been used in wide range of analyses. Feature information was extracted to analyze model structure [6]–[12], and track changes [58]–[60]. Program analysis aimed at finding variability bugs [13]–[15], [61], detecting dead features and inconsistencies between code and model [16]–[19]. For multi-objective optimization [20]–[23], models were used to sample and evaluate configurations to find better performing configurations.

While some `Kconfig` analyses required only parsing `Kconfig` models or using the generated `CNF` formula to check the satisfiability, finding variability bugs and performing multi-objective optimization requires `URS`.

C. Random Sampling Configurations

Randomly selecting features, not configurations, leads to large number of invalid configurations [15], [19]. As SAT solvers had an issue where some features are always selected or unselected, different modifications were made to shuffle variable assignments [21], [23], [62], but did not show true randomness. Dutra *et al.* [63] proposed a random sampling method for test case generation based on MaxSAT solvers, but samples were not uniformly distributed as invalid configurations were sampled as well, which were discarded. Melo *et al.* [61] used randconfig to sample different Linux configurations, which we have shown are not uniformly distributed.

VII. Conclusions

To analyze product spaces and make accurate statistical predictions requires tools. Oh *et al.* [26] provided the ingredients for our research: a theory based on uniform random sampling (URS) that could explore product spaces and tools to generate CNF formulas of feature models which could be analyzed by SAT and BDD tools.

This paper extended prior work to analyze colossal product spaces. We discovered that a key requirement for URS was how CNF formulas of feature models were produced. Prior tools created CNF formulas for their respective needs (finding configurations), but URS demands that formulas should not redundantly encode configurations — an observation that no one had made before.

This led us to develop a pair of tools. Kclause translates Kconfig models into compact CNF formulas that encode each legal configuration once. This was a surprisingly difficult task because the semantics of Kconfig are ambiguous. It took months of experimentation to discover the meaning of different combinations of constraints; even simple specifications had decidedly non-obvious semantics. But we achieved our goal of compactness and non-redundancy, and should our tool encounter a Kconfig model that uses constraints that we have not explored, we warn users. Our experiments showed:

- Kclause produces formulas equivalent to Kconfig models with 95% confidence and 3% margin of error. No other tool is this accurate;
- Sampling solutions of formulas that encode configurations redundantly will not be uniformly random; and
- Kclause formulas are compact, having 63% fewer clauses and 44% faster sampling time on average than formulas produced by existing tools;

Smarch performs URS on Kclause formulas with colossal spaces using #SAT technology. Our experiments:

- Confirmed that an existing tool, Unigen2, could perform URS but could not handle configuration spaces exceeding 10^{13};
- Showed an existing tool, randconfig, cannot be used when URS is needed; and
- Kclause and Smarch yielded accurate statistics on colossal spaces, our largest (*Busybox*) had a titanic size 10^{248}.

Our research also suggests future explorations:

1) Enhance the scalability of Smarch to reduce sampling times, and
2) Evaluate whether URS improves results of prior work.

References

[1] S. Apel, D. Batory, C. Kaestner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[2] "Kconfig tool specification," https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt, 2018.

[3] J. Villanueva, "How many atoms are there in the universe?" https://www.universetoday.com/36302/atoms-in-the-universe/, 2009.

[4] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Handbook of Knowledge Representation*, vol. 3, 2008.

[5] M. Thurley, "sharpsat–counting models with advanced component caching and implicit bcp," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2006, pp. 424–429.

[6] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the linux build system," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 21–30.

[7] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, 2015.

[8] V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann, "Feature models in linux: from symbols to semantics," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2016, pp. 65–72.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[10] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "Sat-based analysis of large real-world feature models is easy," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 91–100.

[11] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: a longitudinal study of linux kernel device drivers," in *Proceedings of the 14th International Conference on Modularity*. ACM, 2015, pp. 81–92.

[12] P. Gazzillo, "Kmax: finding all configurations of kbuild makefiles statically," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 279–290.

[13] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. ACM, 2011, p. 2.

[14] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 421–432.

[15] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 643–654.

[16] S. El-Sharkawy, A. Krafczyk, and K. Schmid, "An empirical study of configuration mismatches in linux," in *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. ACM, 2017, pp. 19–28.

[17] S. Nadi and R. Holt, "Mining kbuild to detect variability anomalies in linux," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 107–116.

[18] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 47–60.

[19] J. Liebig, A. von Rhein, C. Kastner, S. Apel, J. Dorre, and C. Lengauer, "Scalable analysis of variable software," in *ESEC/FSE*, 2013.

[20] J. Chen, V. Nair, R. Krishna, and T. Menzies, ""sampling" as a baseline optimizer for search-based software engineering," *IEEE Transactions on Software Engineering*, 2018.

[21] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 517–528.

[22] J. Guo, J. H. Liang, K. Shi, D. Yang, J. Zhang, K. Czarnecki, V. Ganesh, and H. Yu, "Smtibea: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines," *Software & Systems Modeling*, pp. 1–20, 2017.

[23] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 465–474.

[24] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems," in *ASE*, 2015.

[25] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *ASE*, 2013.

[26] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 61–71.

[27] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[28] S. Nadi and R. Holt, "The linux kernel: A case study of build system variability," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 730–746, 2014.

[29] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "A first step towards a framework for the automated analysis of feature models," 2006, pp. 39–47.

[30] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform sat witness generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 304–319.

[31] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.

[32] Wikipedia, "Conjunctive normal form," https://en.wikipedia.org/wiki/Conjunctive_normal_form, 2018.

[33] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[34] S. Muroga, *Logic Design and Switching Theory*. John Wiley & Sons, Inc., 1979.

[35] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *IEEE, Part I: Communication and Electronics*, vol. 72, no. 5, 1953.

[36] G. S. Tseitin, "On the complexity of derivation in propositional calculus," pp. 466–483, 1983.

[37] P. Jackson and D. Sheridan, "Clause form conversions for boolean circuits," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2004, pp. 183–198.

[38] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293 – 304, 1986.

[39] O. Kullmann, "On a generalization of extended resolution," *Discrete Applied Mathematics*, vol. 96-97, pp. 149 – 176, 1999.

[40] M. Järvisalo, A. Biere, and M. J. H. Heule, "Blocked clause elimination," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 129–144.

[41] "Kconfig language specification," https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt, 2018.

[42] "kconfigreader," https://github.com/ckaestne/kconfigreader, 2016.

[43] DIMACS, "Satisfiability: Suggested format," *DIMACS Challenge. DIMACS*, 1993.

[44] S. El-Sharkawy, A. Krafczyk, and K. Schmid, "Analysing the kconfig semantics and its analysis tools," in *ACM SIGPLAN Notices*, vol. 51, no. 3. ACM, 2015, pp. 45–54.

[45] WolframAlpha, "Boolean algebra simplifier," http://www.wolframalpha.com/input/?i=P+and+not+Q, 2018.

[46] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Haifa Verification Conference*. Springer, 2011, pp. 50–65.

[47] "axtls website," http://axtls.sourceforge.net/, 2018.

[48] "Toybox website," http://landley.net/toybox/, 2018.

[49] "Fiasco website," http://axtls.sourceforge.net/, 2018.

[50] "Busybox website," https://busybox.net/, 2018.

[51] "uclibc-ng website," https://uclibc-ng.org/, 2018.

[52] A. G. Bluman, *Elementary statistics*. Brown Melbourne, 1995.

[53] F. J. M. Jr., "The kolmogorov-smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[54] H. A. David and H. N. Nagaraja, "Order statistics," *Encyclopedia of Statistical Sciences*, vol. 9, 2004.

[55] B. Marker, D. Batory, and R. Van De Geijn, "Understanding performance stairs: Elucidating heuristics," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 301–312.

[56] "Kclause-smarch data archive," https://doi.org/10.5281/zenodo.2574214, 2019.

[57] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 805–824.

[58] N. Dintzner, A. van Deursen, and M. Pinzger, "Analysing the linux kernel feature model changes using fmdiff," *Software & Systems Modeling*, vol. 16, no. 1, pp. 55–76, 2017.

[59] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the linux kernel variability model," in *International Conference on Software Product Lines*. Springer, 2010, pp. 136–150.

[60] A. Ziegler, V. Rothberg, and D. Lohmann, "Analyzing the impact of feature changes in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2016, pp. 25–32.

[61] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2016, pp. 3–8.

[62] J. Chen, V. Nair, R. Krishna, and T. Menzies, ""sampling" as a baseline optimizer for search-based software engineering," *IEEE Transactions on Software Engineering*, 2018.

[63] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 549–559.