Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features

Daniel-Jesus Munoz
CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech
Málaga, Andalucía, Spain
danimg@lcc.uma.es

Jeho Oh
Department of Computer Science
Austin, Texas, USA
jeho@cs.utexas.edu

Mónica Pinto, Lidia Fuentes
CAOSD, Dpt. LCC, Universidad de Málaga, Andalucía Tech
Málaga, Andalucía, Spain
{pinto,lff}@lcc.uma.es

Don Batory
Department of Computer Science
Austin, Texas, USA
batory@cs.utexas.edu

ABSTRACT

Analyses of *Software Product Lines* (SPLs) rely on automated solvers to navigate complex dependencies among features and find legal configurations. Often these analyses do not support numerical features with constraints because propositional formulas use only Boolean variables. Some automated solvers can represent numerical features natively, but are limited in their ability to count and *Uniform Random Sample* (URS) configurations, which are key operations to derive unbiased statistics on configuration spaces.

Bit-blasting is a technique to encode numerical constraints as propositional formulas. We use bit-blasting to encode Boolean and numerical constraints so that we can exploit existing #SAT solvers to count and URS configurations. Compared to state-of-art Satisfiability Modulo Theory and Constraint Programming solvers, our approach has two advantages: 1) faster and more scalable configuration counting and 2) reliable URS of SPL configurations. We also show that our work can be used to extend prior SAT-based SPL analyses to support numerical features and constraints.

KEYWORDS

feature model, bit-blasting, propositional formula, numerical features, model counting, software product lines

ACM Reference Format:
Daniel-Jesus Munoz, 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 - Volume A (SPLC '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3336294.3336297

SPLC '19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-7138-4/19/09...$15.00
https://doi.org/10.1145/3336294.3336297

1 INTRODUCTION

Software Product Lines (SPLs) are highly configurable systems. A *feature model* defines the variability of an SPL using features and constraints. A *feature* is an increment in program functionality. A constraint is a relationship among features, where the presence or absence of some features requires or precludes other features. A valid combination of features is a *configuration*. All configurations define a *configuration space* [2].

Classical feature models use *Boolean features* that have only two values (present, absent). Boolean features are insufficient for real-world SPLs, as there exist features that have a series or a range of numbers as explicit values. An example is the size in bytes of a datafile [49]; it is represented by a power of 10 series of values in a feature model. These features are called *Numerical Features* ($\mathbb{NF}$s). feature models with $\mathbb{NF}$s are *Numerical Feature Models* ($\mathbb{NFM}$s).

It is infeasible to understand large configuration spaces by enumeration. Most SPLs do not have an analytical model to accurately predict run-time properties (*eg.,* [48]), so it is common to sample configurations, build the product for each sample, and gather data about samples by benchmarking. Doing so creates a dataset on the configuration space. This approach has been used many times: deriving the influence of a feature for performance modeling [33, 55], performing multi-objective optimization [26, 34, 35, 56], and evaluating different sampling approaches to locate variability bugs [47, 62].

Counting the number of configurations and *Uniform Random Sampling* (URS) configurations are two operations for unbiased statistical inferences on SPLs. Counting and URS solutions of $\mathbb{NFM}$s, however, are largely unexplored. Only a handful of automated solvers can represent and reason over both Boolean and numerical feature constraints, namely *Satisfiability Modulo Theories* (SMT) [6] and *Constraint Programming* (CP) [53] solvers. Unfortunately, SMT and CP solvers cannot count the number of configurations (except by enumeration) or uniform sample configurations. Prior work sampled configurations with SMT and CP solvers, but whether the produced samples are uniformly distributed was *not* shown.

In contrast, for classical feature models, there are tools that can count faster than SMT and CP solvers and enable URS of configurations. Every classical feature model can be encoded as a propositional formula, where a solution of the propositional formula is a valid configuration of the feature model. *#SAT* solvers extend *Satisfiability* (SAT) solvers to count the number of solutions of a propositional formula without enumeration [12]. Chakraborty et al. and Oh et al. developed tools to URS solutions of a propositional formula, based on #SAT technology [17, 51].

Bit-blasting encodes numerical values as binary bits and represent operations on them as propositional formulas [15]. We propose to represent ℕ𝔽s and their constraints by bit-blasting and utilize existing SAT-based tools for counting and URS classical feature models. We make use of the `'Tactic'` functionality of the Z3 SMT solver [22] to convert ℕ𝔽s and their constraints into propositional formulas using bit-blasting, which are then integrated with the propositional formulas of classical feature models. This allows us to represent ℕ𝔽𝕄s as a *Bit-Blasted Propositional Formula* (𝔹𝔹ℙ𝔽).

𝔹𝔹ℙ𝔽 can be input to existing #SAT-based tools for counting and URS solutions of a ℕ𝔽𝕄, which SMT and CP solvers cannot do. In this way ℕ𝔽𝕄s can be analyzed by existing tools with minimal extra work. The contributions of our work are:

- Use of bit-blasting to express ℕ𝔽s and constraints,
- Integration of bit-blasting and classical feature models to translate a ℕ𝔽𝕄 into a 𝔹𝔹ℙ𝔽,
- Experiments that show counting and URS solutions of 𝔹𝔹ℙ𝔽 outperform SMT and CP solvers, and
- Evaluation of known SPL analyses using ℕ𝔽𝕄s with huge configuration spaces, the largest exceeding 10^{45} products.

2 BACKGROUND

2.1 Bit-Blasting

Bit-blasting or *flattening* is the transformation of a bit-vector arithmetic formula to an equivalent propositional formula [3]. It has been mainly used in hardware verification [19] and to optimize the hardware verification task itself [27, 66]. Brillout *et al.* [13] used bit-blasting to create a bit-accurate and complete decision procedure for IEEE-compliant binary floating-point arithmetic units.

We focus on the following arithmetic operations: equality (=), inequalities (≠, >, ≥), addition (+) and subtraction (−). Although bit-blasting supports more operations, it is known that multiplication and division do not scale with increasing bit-width [15]. Real-world SPLs that we have studied are described in Table 2. They largely limit their use of numerical operations to equality and inequalities. A few add two ℕ𝔽s and compare the result to a constant. Technical details on bit-blasting are covered later in Section 3.

2.2 Feature Models

A classical feature model uses only Boolean features but this very restriction allows it to be transformed into a propositional formula, where features are variables and constraints are clauses [2]. Many tools can convert an feature model into a propositional formula. One is FeatureIDE that exports a feature model written in their tool as a *Conjunctive Normal Form* (CNF) formula [63]. Another is KClause which transforms a KConfig model into a compact CNF formula [38].

Real-world SPLs use ℕ𝔽𝕄s that contain both binary features and ℕ𝔽s [36]. An ℕ𝔽 has a name N, a type (*ie.,* domain), and range (*eg.,* $N \in [1, 2, \ldots 128]$). ℕ𝔽𝕄s add new constraints to the set of propositional connectives, including: numerical equality (=), numerical inequalities (≠, >, <, ≥ and ≤), and occasionally addition and subtraction but no other numerical operations (at least in KConfig systems [28, 29]). ℕ𝔽s can also have constraints with Boolean features, where the value of an ℕ𝔽 affects the value of a Boolean feature, and vice versa.

Two examples of ℕ𝔽𝕄s are: (1) the HADAS eco-assistant [48] where energy context parameters are represented as ℕ𝔽s in an Integer domain, and propositional connectives and inequalities are present in cross-tree constraints (*eg.,* $AES\_crypto \Rightarrow key\_size > 128$) and (2) WeaFQAs [37] where some variables of quality attributes are ℕ𝔽s with Integer or Float domains, containing propositional connectives and interval constraints (*ie.,* numerical value ranges).

2.3 Uniform Random Sampling and Finding Sub-Optimal Products in Colossal Spaces

Uniform sampling ensures all samples are valid and uniformly distributed across the configuration space, so that the samples can be used for standard statistic approaches.

Oh *et al.* [51] were the first to URS an SPL configuration space. They used the following ideas: Let ϕ be the propositional formula of a classical feature model. Let $S(\phi)$ be the set of all solutions of ϕ. Each solution of ϕ is in a 1-to-1 correspondence with a configuration product in the feature model [2].

Let $|S(\phi)|$ be the number of solutions in $S(\phi)$. A uniform random number generator can select an integer j in the range $[1..|S(\phi)|]$. The trick is to convert j into the j^{th} configuration in a fixed linear ordering of $S(\phi)$. By construction, URS of numbers in $[1..|S(\phi)|]$ is isomorphic to URS of configurations in $S(\phi)$.

SAT solvers find solutions to a given ϕ. #SAT solvers, a relatively new SAT technology, can count $|S(\phi)|$. And #SAT can also be used to convert an integer j into the j^{th} configuration of $S(\phi)$ [38]. Here's how: Let $\mathbb{F}=(f_1, f_2, \ldots)$ be a fixed-order list of all features in a feature model. A #SAT tool can count $n=|S(\phi \wedge f_1)|$, the number of solutions that have feature f_1. If $j \leq n$, then the j^{th} configuration must have feature f_1 otherwise it has $\neg f_1$. Repeating this logic on the remaining features in $\mathbb{F}$ performs a binary search on $S(\phi)$ to reveal the presence or absence of every feature in the j^{th} configuration.

Here is a great application for URS: it can be used to quickly locate sub-optimal products in $S(\phi)$. Take n URS in $S(\phi)$, build and benchmark each of them. Let c_{best} be the best performing configuration among these n samples. Oh *et al.* [51] showed that c_{best} will be, on average, within the top $\frac{1}{n+1}$ percentile of the best performing configurations in $S(\phi)$. So if 99 uniformly random samples are taken, c_{best} is in the top 1% of the best performing configurations of $S(\phi)$, on average, *no matter how big* $|S(\phi)|$ *is* [51]. We explore this application further on Section 4.

3 BIT-BLASTING FOR ℕ𝔽𝕄S

We describe how to integrate bit-blasting and classical feature models to form ℕ𝔽𝕄s and how to translate a ℕ𝔽𝕄 into a 𝔹𝔹ℙ𝔽.

3.1 Bit-Blasting for Arithmetic Operations

This section reviews ideas about bit-blasting that are known to be implemented by Z3. Bit-vectors have two properties: width of the vector and whether it is unsigned (binary *sign-magnitude* encoding)

#	Operation	Bit-Blasted Model	Propositional Formula
1	$(NF_a == NF_b)$	$(a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 == b_1)$	$(a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_1 \Leftrightarrow b_1)$
2	$(NF_a \neq NF_b)$	$(a_3 \neq b_3) \vee (a_2 \neq b_2) \vee (a_1 \neq b_1)$	$(a_3 \oplus b_3) \vee (a_2 \oplus b_2) \vee (a_1 \oplus b_1)$
3	$(NF_a > NF_b)$	$(a_3 < b_3) \vee ((a_3 == b_3) \wedge (a_2 > b_2)) \vee$ $((a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 > b_1))$	$(\neg a_3 \wedge b_3) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \wedge \neg b_2)) \vee$ $((a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_1 \wedge \neg b_1))$
4	$(NF_a \geq NF_b)$	$(a_3 < b_3) \vee ((a_3 == b_3) \wedge (a_2 \geq b_2)) \vee$ $((a_3 == b_3) \wedge (a_2 == b_2) \wedge (a_1 \geq b_1))$	$(\neg a_3 \wedge b_3) \vee ((a_3 \Leftrightarrow b_3) \wedge (b_2 \Rightarrow a_2)) \vee$ $((a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (b_1 \Rightarrow a_1))$
5	$(NF_a \pm NF_b)$	$S_1^4 \equiv [(a_1 \oplus b_1) \oplus C_0, (a_2 \oplus b_2) \oplus C_1,$ $(a_3 \oplus b_3) \oplus C_2, C_3]$ $C_1^3 \equiv (a_i \wedge b_i) \vee C_{i-1}$ $C_0 \equiv (`+' \Rightarrow 0) \wedge (`-' \Rightarrow 1)$	$[(a_1 \oplus b_1) \oplus \pm, (a_2 \oplus b_2) \oplus ((a_1 \wedge b_1) \vee \pm),$ $(a_3 \oplus b_3) \oplus ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm)),$ $(a_3 \wedge b_3) \vee ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm))]$

Table 1: Bit-blasted Models and propositional formula Transformation Examples for 2-bit two's Complement Signed Integers

or signed (binary *two's complement* [1] encoding). We use the Little-Endian representation, *ie.,* signed bit-vectors, where the last bit encodes the sign as positive (0) or negative (1).[2]

Table 1 shows examples of two's compliment bit-blasting propositional formulas for equality, inequality, greater, greater or equal, and addition/subtraction of Little-Endian signed integers with a value range of [-4, 3] (*ie.,* $n = 3$ bits) where a_3 is the integer sign. Of course, a greater number of bits can be used in Table 1, but n=3 shows the repeating patterns in propositional formula that bit-blasting uses. Equality (==) is the conjunction of bit-by-bit equivalences (row 1, col propositional formula). Inequality ($\neq$) is a bit-by-bit disjunction of logical *XORs* ($\oplus$) (row 2, col propositional formula). After the numerical sign comparison (first clause of col PF in rows 3 and 4), there are bit-by-bit equivalences till the last bit of the series, which involve an implication in case of $\geq$ (row 4, col 3), or a disjunction of opposites in case of > (row 3, col 3).

Bit-blasting addition is harder. Addition of bit-vectors can create a result outside the range of the operands due to the number of bits necessary to represent the result. For example, for 3 signed bits, if we perform '3 + 1', the result is '4', which is impossible to represent with 3 signed bits; we need 4 signed bits. The extra bit is called a *carry bit*. Then, a binary addition requires two data inputs, and produces two outputs, the Sum (S) of the equation and a Carry (C) bit as shown in the operation 5 of Table 1. Subtraction in a two's complement encoding is an addition differing on C_0, which is 0 for addition operations, and 1 for subtraction operations.

SAT solvers regularly work with propositional formulas in CNF form [12]. To transform the propositional formulas of Table 1, the Z3 solver uses Tseitin's CNF transformations with skolemization [65], as it is a widely known method to transform propositional formulas into a CNF formula while maintaining the model satisfiability and number of configurations.

3.2 Producing a $\mathbb{BBPF}$ for an $\mathbb{NFM}$

We encode the Boolean features and their constraints of an $\mathbb{NFM}$ as a propositional formula in the standard way [2]. Then, $\mathbb{NF}$s and their constraints of a $\mathbb{NFM}$ are encoded as propositional clauses making use of the Z3 solver `'Tactic'` functionality.[3] We conjoin both predicates (or substitute them below) to form the $\mathbb{BBPF}$ for that $\mathbb{NFM}$. Here are some details:

$\mathbb{NF}$ **Definition.** Let a signed $\mathbb{NF}$ f have range $[a, b]$. Bit-blasting uses $\lceil \log_2(max(|a|, |b|)) + 1 \rceil + 1$ variables to represent the bits of f, where 1 variable encodes the sign. Propositional clauses for two constraints $(f \geq a)$ and $(f \leq b)$ are conjoined to limit the range of f values. If applicable, the range of f is shifted to $[0, b - a]$ as it may simplify the formula and use fewer bits, namely $\lceil \log_2(b - a) + 1 \rceil + 1$.

We represent all $\mathbb{NF}$s as integers. Decimal point values can be represented by shifting the points to the desired precision, which is shifted back when the configurations are sampled.

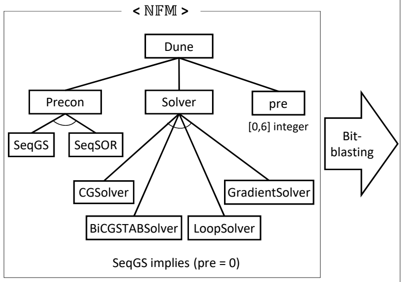
$\mathbb{NF}$ **Constraints.** Constraints between $\mathbb{NF}$s can be directly derived as propositional clauses from bit-blasting and conjoined to a propositional formula. If an $\mathbb{NF}$ is a constant, its binary value is used, which can simplify the formula by Boolean constraint propagation.

Two $\mathbb{NF}$s bounded under the same constraint may have different bit-widths due to different value ranges. As the bit-width of each $\mathbb{NF}$ is fixed, the $\mathbb{NF}$ with shorter bit-width needs to be extended to match the bit-width of the other $\mathbb{NF}$. Extending the bit-width does not change an $\mathbb{NF}$'s possible values due to range constraints.

Mixed Boolean and $\mathbb{NF}$ **Constraints.** A numerical constraint can be qualified by Boolean features, such as $a \Rightarrow (b \neq 0)$, where a is a

[1]Two's complement negative integer encoding is the binary complement of the positive encoding plus one bit.
[2]Little-Endian: An order of bits in which the "little end" (least significant value in the sequence) is represented first in the sequence.

[3]We have configured Z3 solver to convert $\mathbb{NF}$s and constraints into propositional formulas with the *'Tactic'* functionality command `<Then('simplify', 'bit-blast', 'tseitin-cnf')>` However, Z3 solver is not primarily intended to be used for this task, nor it is a well-documented functionality.




```
1  𝔽w𝔹 =  (Precon) ∧ (Solver) // Precon and Solver are mandatory features

2      ∧ ((Precon ⇒ (SeqGS ∨ SeqSOR)) ∧ (SeqGS ⇒ ¬SeqSOR)) // SeqGS and SeqSOR are alternative features of Precon

3      ∧ ((Solver ⇒ (CGSolver ∨ BiCGSTABSolver ∨ LoopSolver ∨ GradientSolver))
         ∧ (CGSolver ⇒ ¬BiCGSTABSolver) ∧ (CGSolver ⇒ ¬LoopSolver) ∧ (CGSolver ⇒ ¬GradientSolver)
         ∧ (BiCGSTABSolver ⇒ ¬LoopSolver) ∧ (BiCGSTABSolver ⇒ ¬GradientSolver) ∧ (LoopSolver ⇒ ¬GradientSolver))
         // CGSolver, BiCGSTABSolver, LoopSolver, and GradientSolver are alternative featrues of Solver

4      ∧ ((¬pre_4 ∧ false) ∨ ((pre_4 ⇔ false) ∧ (false ⇒ pre_3))
         ∨ ((pre_4 ⇔ false) ∧ (pre_3 ⇔ false) ∧ (false ⇒ pre_2))
         ∨ ((pre_4 ⇔ false) ∧ (pre_3 ⇔ false) ∧ (pre_2 ⇔ false) ∧ (false ⇒ pre_1))) // pre >= 0

5      ∧ ((¬false ∧ pre_4) ∨ ((false ⇔ pre_4) ∧ (pre_3 ⇒ true))
         ∨ ((false ⇔ pre_4) ∧ (true ⇔ pre_3) ∧ (pre_2 ⇒ true))
         ∨ ((false ⇔ pre_4) ∧ (true ⇔ pre_3) ∧ (true ⇔ pre_2) ∧ (pre_1 ⇒ false))) // 6 >= pre

6      ∧ (SeqGS ⇒ ((pre_4 ⇔ false) ∧ (pre_3 ⇔ false) ∧ (pre_2 ⇔ false) ∧ (pre_1 ⇔ false)))
         // SeqGS implies (pre == 0)
```

Figure 1: An example of ℕ𝔽𝕄 to 𝔹𝔹ℙ𝔽 Conversion

Boolean feature and b is a ℕ𝔽. In this case, the propositional clauses for ℕ𝔽 operations can be generated first (*eg.,* let ω be the bit-blasted propositional formula of $(b \neq 0)$), which is then substituted into the original formula to yield the result, namely $a \Rightarrow \omega$.

A constraint may inhibit a ℕ𝔽 from having any value, meaning that the ℕ𝔽 is not used and its value is ignored. In such case, a designated value outside the range of the ℕ𝔽 can be used to indicate the ℕ𝔽 is ignored, enforced by an equals operation.

Alternative Features. For a large set of alternative features, representing them as an ℕ𝔽 and keeping a map between its values and alternative features may derive a more compact propositional formula. As an extreme case, 2^n alternative features require 2^n variables, while representing them as a single ℕ𝔽 requires only n bits. Regarding the clauses, alternative features requires $\binom{2^n}{2}+1$ CNF clauses,[4] while an ℕ𝔽 requires none. A ℕ𝔽 that allows multiple discrete values (*eg.,* odd numbers $\{2, 3, 7, 11, 13...\}$) instead of values within a range can be encoded in the same manner.

Fig. 1 shows our encoding of an ℕ𝔽𝕄 as a 𝔹𝔹ℙ𝔽. This ℕ𝔽𝕄 was taken from the Dune multi-grid solver [32]. Note that some features and constraints are modified for better illustration.

In Fig. 1, the clauses for Boolean features are represented in lines 1–3, while the clauses for ℕ𝔽 is conjoined at lines 4–6. As the ℕ𝔽 '`pre`' has range [0, 6], 4 bits are allocated (including '`pre_4`' as its sign bit). Lines 4 and 5 specify the range of the '`pre`' feature. Line 6 encodes the constraint between a Boolean feature and a ℕ𝔽, where bit-blasting clauses for an equality operation has an implication relationship with a Boolean feature '`SeqGS`'.

4 EVALUATION

Our work counts and uniform samples configurations of ℕ𝔽𝕄s. We answer the following research questions to evaluate 𝔹𝔹ℙ𝔽:

RQ1 — How many bits per ℕ𝔽 are feasible with bit blasting (𝔹𝔹)?
RQ2 — Does 𝔹𝔹ℙ𝔽 allow faster counting?
RQ3 — Does 𝔹𝔹ℙ𝔽 allow URS?
RQ4 — Can existing SAT-analyses of SPLs use 𝔹𝔹ℙ𝔽?

RQ1 evaluates a scalability metric of bit blasting, while **RQ2** and **RQ3** evaluate how 𝔹𝔹ℙ𝔽 perform compared to state-of-art SMT and CP solvers. **RQ4** evaluates whether 𝔹𝔹ℙ𝔽 can be used with existing SAT-analyses for SPLs.

We used real-world ℕ𝔽𝕄s from [38] and [58] that constrain both Boolean and numerical features. Table 2 lists each ℕ𝔽𝕄 with its description, where each system has a different number of ℕ𝔽s and/or difference configuration space size. Henceforth, we use FSE2015 to denote the feature models from Siegmund *et al.* [58].

FSE2015 ℕ𝔽𝕄s have relatively small configuration spaces, but the equation solving times of all the configurations were benchmarked, so that we can rank them. These ℕ𝔽𝕄s were written for the SPLConqueror tool [58], which we have translated into 𝔹𝔹ℙ𝔽. Their smallest ℕ𝔽𝕄 had range [1,4]; the largest had [66,4096].

Compared to FSE2015 ℕ𝔽𝕄s, KConfig models have many more features and have huge configuration spaces. The KClause tool [58] derives a propositional formula for each KConfig model. As KClause simplifies ℕ𝔽s to have their default values only, we augmented their formula with bit blasting to allow different ℕ𝔽 values.

The KConfig ℕ𝔽𝕄s that we examined had ℕ𝔽s as small as [0,1] and as large as $[0, 2^{32}-1]$. When the range of a ℕ𝔽 is not defined in a KConfig model, any value within the range of the integer data type is possible, which is $[0, 2^{32}-1]$. For ℕ𝔽s that exceed the range $[0, 2^{10}-1]$, we discretized them to have 2^{10} possible values. We benchmarked the build size of each sampled configuration for the performance analysis of **RQ4**.

To generate a propositional formula for an ℕ𝔽 and constraints for 𝔹𝔹ℙ𝔽, we used the formula printing functionality of the Z3 solver. To count the number of configurations in 𝔹𝔹ℙ𝔽, we used sharpSAT [64], a state-of-art model counter for propositional formulas. To sample configurations of a 𝔹𝔹ℙ𝔽, we used Smarch [38], a state-of-art tool for URS propositional formula solutions.

RQ1: How many bits per ℕ𝔽 are feasible with 𝔹𝔹?

The most complicated numerical constraint that appeared in the systems we analyzed is $(A+B>C)$, where A, B, and C are ℕ𝔽s of unsigned integers. We consider this constraint as an upper bound on the overhead of numerical constraints.[5] Propositional formulas with three b-bit ℕ𝔽s related by this constraint were generated and benchmarked to determine how many bits are feasible for counting and URS.

Formulas with different bit-widths (#b) from 2 to 32 were generated. For each formula, we measured:

[4] $\binom{2^n}{2}$ clauses are need to ensure only one among the alternative feature is selected, while another clause is to ensure at least one among them is selected.

[5] Actual numerical constraints were simpler, as C was substituted with a constant. Constants in constraints simplifies the formula by Boolean constraint propagation.

Type	$\mathbb{NFM}$	**Description**	**# Features**	**# $\mathbb{NF}$s**	**# Configs**	**Benchmark**	**Ref.**
FSE2015	Dune	Multi-grid solver	11	3	$2,304$	Equation solving times	[58]
	HSMGP	Stencil-grid solver	14	3	$3,456$		
	HiPAcc	Image processing framework	33	2	$13,485$		
	Trimesh	Triangle mesh library	13	4	$239,360$		
KConfig	axTLS	Client-server library	94	9	4.96×10^{38}	Build sizes	[38]
	Fiasco	Real-time microkernel	234	5	3.06×10^{12}		
	uClibc-ng	C library for embedded Linux	269	6	8.20×10^{45}		

Table 2: List of Models with Numerical Features and Constraints

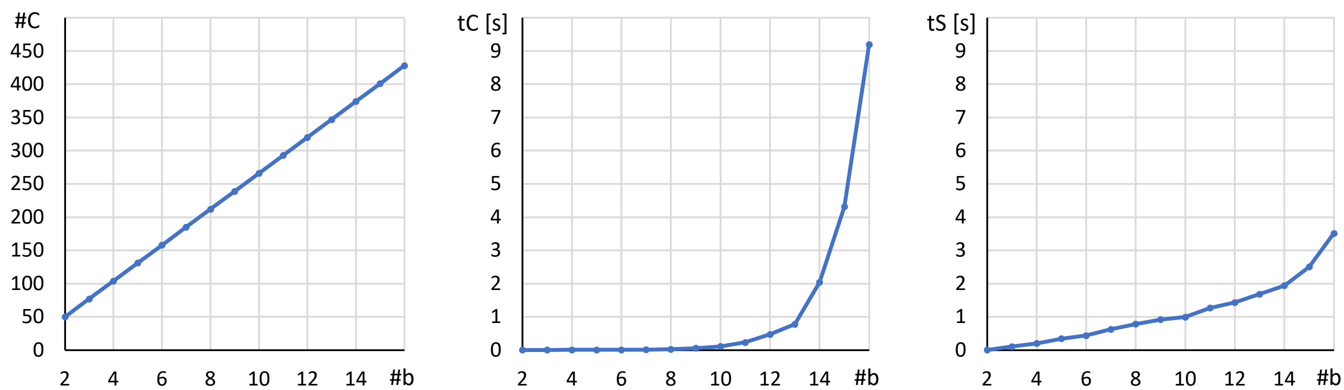


Figure 2: Computation Overhead of the $(A + B > C)$ Constraint

$\#C$ — number of CNF clauses in each formula,
tC — time in seconds to count configurations by sharpSAT, and
tS — time in seconds to sample a configuration by Smarch.

To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. If counting or URS took more than 10 seconds, we considered it a *time-out*.

Figure 2 shows our results. The Y-axes show $\#C$, tC, and tS; the X-axes are the number of bits ($\#b$). As tC timed-out after 16 bits, we show $\#b$ up to 16. We observed:

- $\#C$ grew linearly with increasing $\#b$,
- tC grew exponentially with increasing $\#b$, tC was below 1 second for $\#b \leq 13$,
- tS grew exponentially with increasing $\#b$, and tS was below 1 second for $\#b \leq 10$.

The linear increase of $\#C$ is due to the use of Tseitin's transformation in generating CNF formulas. As the number of $\mathbb{NF}$ variables increases with linearly with $\#b$, Tseitin's transformation guarantees a linear increase $O(3n+1)$ with the number of variables [65].

tS showed a slower rate of increase compared to tC, so $tC > tS$ from $\#b > 14$. This is due to the formula partitioning of Smarch, which made counting solutions to large formulas faster by counting in a divide-and-conquer manner [38].

These results give a rough idea of the overhead added by $\mathbb{NF}$s with constraints. The fact that there was a time-out after 16 bits does not mean that $\mathbb{NF}$s larger than 16 bits cannot be treated by bit blasting. When a $\mathbb{NF}$ has a value requiring more than 16 bits, we can discretize it to reduce the number of bits to encode it. For example, a 32-bit $\mathbb{NF}$ of range $[0, 2^{32}-1]$ can be discretized into a 10-bit $\mathbb{NF}$ with the precision of 2^{22}. This makes analyses feasible by reducing the precision of possible values.

Conclusion: Bit-blasting is feasible up to 16 bits per number (<10 seconds) and has negligible overhead up to 10 bits per number (<1 second).

RQ2: Does $\mathbb{BBPF}$ allow faster counting?

We compared the time to count solutions using sharpSAT and widely-used SMT and CP solvers. We used Z3[6] as a representative SMT solver and Clafer with the Choco solver[7] as a representative CP solver. Z3 and Clafer use different ways to count the number of configurations than sharpSAT:

- Z3 does not have the functionality to count configurations. A known method involves enumerating the configurations by: 1) deriving a configuration from Z3, 2) making the negation of that solution as a constraint, and 3) repeating 1) and 2) until the constrained model is not satisfiable.[8]
- Clafer has an internal functionality to count configurations, by using the option '-n'. Its functionality involves enumerating configurations as well.[9]

We measured the time in seconds to count configurations by each tool. To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10%

[6] Z3py, https://github.com/Z3Prover/z3.
[7] Clafer, https://www.clafer.org/.
[8] Z3 repository developer response on model counting, https://github.com/Z3Prover/z3/issues/934.
[9] Clafer Choco solver source code, https://github.com/chocoteam/choco-solver/blob/master/src/main/java/org/chocosolver/solver/search/strategy/Search.java.

Type	Model	Z3	Clafer	𝔹𝔹ℙ𝔽
FSE2015	Dune	26.20s	11s	0.01s
	HSMGP	40.70s	14s	0.01s
	HiPAcc	458s	33s	0.01s
	Trimesh	Time-out	2s	0.01s
KConfig	axTLS	Time-out	Time-out	0.01s
	Fiasco	Time-out	Time-out	0.01s
	uClibc-ng	Time-out	Time-out	0.01s

Table 3: Average Counting Time Comparison

margin of error [61]. If counting took more than 30 minutes, we considered it a *time-out*. Table 3 shows our results.

We observed with 𝔹𝔹ℙ𝔽:

- As expected, counting 𝔹𝔹ℙ𝔽 by sharpSAT was much faster than Z3 and Clafer, as it does not enumerate solutions.
- KConfig ℕ𝔽𝕄s are too large for Z3 and Clafer to enumerate.

Conclusion: SharpSAT with 𝔹𝔹ℙ𝔽 counts configurations considerably faster than Z3 and Clafer. Z3 and Clafer were unusable for KConfig models.

RQ3: Does 𝔹𝔹ℙ𝔽 allow URS?

We now ask if 𝔹𝔹ℙ𝔽 with Smarch, Z3, and Clafer can URS solutions of a ℕ𝔽𝕄. **RQ2** showed Z3 and Clafer can generate samples by enumeration but did not reveal if their samples are uniform. We used techniques in prior work to obtain random samples:

- For Z3, we randomly assigned the value for the parameter 'random_seed', which controls the variable selection heuristic [34].
- For Clafer, we set the '–search' option to 'random', which randomizes the order and value of variable assignments.[9]

To check if the samples are uniformly distributed, we rely on a theorem from [38, 51]. Order statistics predict that the average rank of samples from URS are evenly distributed across a configuration space. So if n samples are taken, the configuration space is partitioned into n+1 equal-length intervals *on average*. The normalized rank (in the unit interval) of the k^{th}-best performing sample is $\frac{k}{(n+1)}$ [51].

With this result, we can check whether samples have evenly distributed ranks using the *Kolmogorov-Smirnov* (KS) test [45]. A KS test checks whether two data sets are sampled from the same distribution. We check if the distribution between sampled ranks and expected ranks are equal with 95% confidence.

First, we used the four ℕ𝔽𝕄s from FSE2015. These ℕ𝔽𝕄s had all of their configurations enumerated and benchmarked, allowing us to know the exact rank of the samples. For each FSE2015 ℕ𝔽𝕄 and each tool, we sampled 100, 300, and 500 configurations to evaluate randomness with different sample sizes. For each sample set, we derived the KS test result and the time taken to sample a configuration, averaged from the sample set, in seconds.

FSE2015 Systems. Rows 1 through 4 in Table 4 (next page) show the average time taken to sample a configuration for each FSE2015 ℕ𝔽𝕄. Table 5 (next page) shows the result of KS test. We observed:

- Z3 and Clafer had fast average sampling times at .03 and .01 seconds; Smarch took more time at .30 secs,
- Smarch passed KS tests for all ℕ𝔽𝕄s and sample sizes, which says that Smarch performs URS with 95% confidence, and
- Z3 and Clafer failed KS tests for some ℕ𝔽𝕄s and sample sizes. This says that the randomization options for Z3 and Clafer do not always achieve URS.

It is unclear what characteristics of a ℕ𝔽𝕄 causes Z3 and Clafer samples to be biased. Prior work on feature models with only Boolean features tried a similar approach to produce random solutions using SAT solvers [18, 35], but they too did not demonstrate URS. In contrast, Smarch delivers URS by construction. That is, it creates a 1-to-1 mapping between a random number and a unique configuration via counting. With Z3 and Clafer, counting is infeasible, as **RQ2** showed.

KConfig Systems. To demonstrate scalability of sampling, we also present the evaluation with KConfig ℕ𝔽𝕄s. Note that, we could not check the randomness of the samples in the same manner a FSE2015 ℕ𝔽𝕄s, as their configuration space cannot be enumerated to obtain the precise rank of selected configurations.[10] Instead, we utilized the evaluation method in [38], to evaluate whether samples from Z3 and Clafer are uniformly distributed using Smarch.

Smarch achieves URS by using a one-to-one mapping between a number and a configuration. When a random number between 1 to the total number of configurations is given, Smarch outputs a corresponding configuration. Smarch also is capable of the inverse operation, so that it outputs the corresponding number of a given configuration. For the samples taken from Z3 and Clafer, we used Smarch to output the numbers and consider them as the rank of those samples. We then evaluated whether those ranks are uniformly distributed using the KS test.

Rows 5 through 7 in Table 4 and Table 5 are the results. KConfig systems shows a similar trend with the FSE2015 systems. Even for models with larger ranged ℕ𝔽s and larger configuration spaces, Smarch was able to sample configurations within a reasonable time. Samples from Z3 and Clafer failed the KS test for all systems and sample sizes, indicating that these tools are not capable of URS configuration spaces that are large as KConfig. On the other hand, samples from Smarch passed all KS tests, as it used uniform random numbers to generate the configurations.

Conclusion: Smarch can perform URS of ℕ𝔽𝕄 configurations, which Z3 and Clafer cannot guarantee.

RQ4: Can existing SAT-analyses of SPLs use 𝔹𝔹ℙ𝔽s?

We explained in Section 2.3 how URS can help finding near-optimal configurations. (Recall taking n samples, benchmarking each selected configuration, and identifying c_{best} — the best performing configuration, would be in the top $\frac{1}{(n+1)}$ percentile of all configurations *on average*.) Oh *et al.* [51] also proposed a recursive searching algorithm called SRS, which recursively: 1) samples configurations, 2) use samples to reason features that improves performance, and 3) constricts the search space with the found features. They demonstrated that SRS performs better than URS alone.

[10] We could sort all FSE2015 configurations by performance, so finding the performance rank of a given configuration is easy. In KConfig systems, we do **not** know the performance of all configurations – only those that we sample. Hence we can only estimate the performance rank of a given configuration.

	$\mathbb{NFM}$	Z3			Clafer			Smarch+$\mathbb{BBPF}$		
Type	*# Samples*	**100**	**300**	**500**	**100**	**300**	**500**	**100**	**300**	**500**
FSE2015	**Dune**	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.25	0.25	0.28
	HSMGP	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.30	0.31	0.31
	HiPAcc	0.05	0.05	0.05	<0.01	<0.01	<0.01	0.54	0.54	0.55
	Trimesh	0.02	0.02	0.02	<0.01	<0.01	<0.01	0.60	0.61	0.61
KConfig	**axTLS**	0.12	0.12	0.12	0.02	0.01	0.01	1.26	1.30	1.27
	Fiasco	0.25	0.24	0.24	0.03	0.01	0.01	1.50	1.49	1.51
	uClibc-ng	0.28	0.27	0.27	0.05	0.02	0.02	5.56	5.39	5.62

Table 4: Average Sampling Times Comparison

	$\mathbb{NFM}$	Z3			Clafer			Smarch+$\mathbb{BBPF}$		
	# Samples	**100**	**300**	**500**	**100**	**300**	**500**	**100**	**300**	**500**
FSE2015	**Dune**	**Fail**	Pass	**Fail**	Pass	Pass	Pass	Pass	Pass	Pass
	HSMGP	Pass	**Fail**	Pass	Pass	Pass	**Fail**	Pass	Pass	Pass
	HiPAcc	**Fail**	**Fail**	**Fail**	Pass	Pass	Pass	Pass	Pass	Pass
	Trimesh	**Fail**	**Fail**	**Fail**	Pass	**Fail**	**Fail**	Pass	Pass	Pass
KConfig	**axTLS**	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	Pass	Pass	Pass
	Fiasco	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	Pass	Pass	Pass
	uClibc-ng	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	**Fail**	Pass	Pass	Pass

Table 5: KS Test Result on Uniformity of Samples

Their work, however, focused on feature models with Boolean features and constraints, while optimizing feature models with $\mathbb{NF}$s and constraints was left as future work. We wanted to see if their work could be used "as is" with $\mathbb{BBPF}$.

We replicated SRS to find near-optimal configurations from the $\mathbb{NFM}$s of **RQ2**. For FSE2015 $\mathbb{NFM}$s, we tried to find the configuration with the smallest benchmarked performance, which was the "equation solving time" (see [58] for details). For an experiment, we performed SRS with 20 samples per recursion, which was claimed in [51] as a sufficient sample size to make accurate statistical decisions on the features.

From the FSE2015 $\mathbb{NFM}$s, we gathered following metrics per experiment regarding configuration ranks:

- *N* — total number of samples taken by SRS,
- *rSRS* — normalized rank of c_{best}, found from SRS,
- *rURS* — expected normalized rank of c_{best} from N configurations by URS. It is derived from order statistics, as $\frac{1}{(N+1)}$.

In addition, we analyzed the performance value of the found configurations from FSE2015 $\mathbb{NFM}$s. Performance values were normalized by the actual best and worst performance value in the configuration space. We measured:

- *pSRS* — normalized performance of c_{best} found by SRS, and
- *pURS* — normalized performance of the configuration at rank *rURS*.

To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. From the experiments. we derived:

- *rTest* — Mann-Whitney U test results, which evaluates whether *rSRS* values are smaller than *rURS* values with 95% confidence [44]. "Pass" implies *rSRS* is smaller, and "Fail" otherwise.[11]
- *pTEST* — Mann-Whitney U test results from 97 experiments which evaluates whether *pSRS* values are smaller than *pURS* values with 95% confidence [44]. "Pass" implies *pSRS* is smaller, and "Fail" otherwise.
- *rBetter* — SRS success rate is the percentage of experiments that SRS outperforms URS, where *rSRS*<*rURS* is expected.

Table 6 shows the rank results for each FSE2015 $\mathbb{NFM}$. We observed:

- The average rank of solutions SRS found were ~.8% away from optimal; the average rank of solutions URS found were ~1.4% away from optimal. Both are good results.
- *N* was different for all $\mathbb{NFM}$s, as the number of features, constraints, and how a feature affects the objective to optimize are different for each $\mathbb{NFM}$,
- *rSRS* was lower than *rURS* for all $\mathbb{NFM}$s with 95% confidence, which indicates SRS outperforms URS,
- *pSRS* was lower than *pURS* for all $\mathbb{NFM}$s with 95% confidence as well, which also indicates SRS finds better performing configurations than URS, and
- *rBetter* was not 100% in all experiments, meaning that occasionally SRS performs worse than URS. SRS performs better than URS in 89% of all the experiments.

To visualize our results, Figure 3 plots all the configurations of the FSE2015 $\mathbb{NFM}$s, sorted by their performance. The X-axis denotes their normalized rank, while the Y-axis denotes their normalized performance. The red dot (●) indicates the configuration found from SRS, while the black × symbol indicates the expected configuration from URS.

Figure 3 shows the configuration found from both SRS and URS are very close to the actual best configuration, regarding both X and Y axis. One exception is Dune, where the best two configurations had much better performance compared to all others. SRS was

[11] We used Mann-Whitney U Test as the distributions of the results are non-parametric.

$\mathbb{NFM}$	***N***	***rSRS***	***rURS***	***rTest***	***pSRS***	***pURS***	***pTest***	***rBetter***
Dune	71.32	0.007	0.016	Pass	0.039	0.042	Pass	93%
HSMGP	66.42	0.008	0.017	Pass	0.005	0.011	Pass	91%
HiPAcc	65.82	0.010	0.017	Pass	0.002	0.004	Pass	82%
Trimesh	129.21	0.003	0.009	Pass	0.003	0.013	Pass	91%

Table 6: Finding Near-Optimal Configurations for FSE2015 Systems

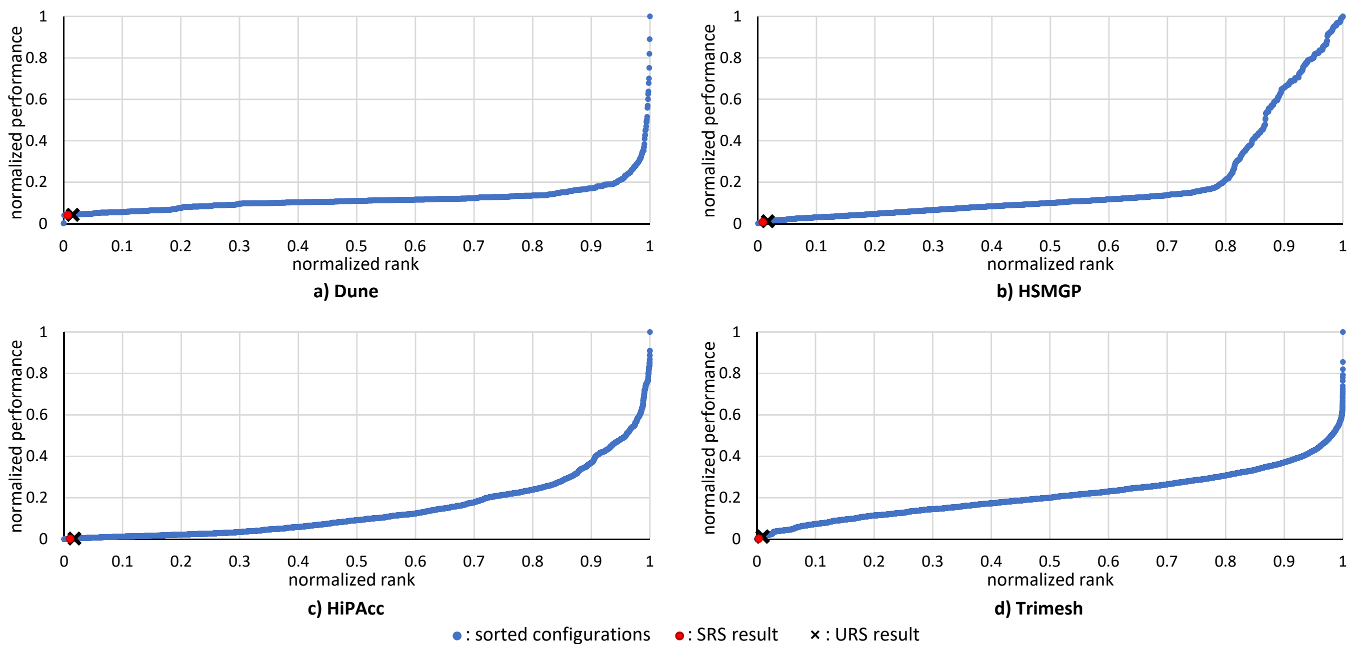


Figure 3: Configuration Space of FSE2015 Systems

able to find the two configurations for some experiments, but not always.

We performed similar experiments with KConfig $\mathbb{NFM}$s to find configurations with the smallest build size. Since the configuration space cannot be enumerated, we do not know what the best and worst performance values are as well as the rank of the returned configurations. At least, we compared the non-normalized build size of configurations found from SRS to that of URS. To do so, we limited N to 200 and derived:

pSRS — smallest build size in megabytes, found from SRS with 200 samples, and

pURS — smallest build size in megabytes, found from the configurations in 200 URS.

We repeated the experiment 25 times and averaged the result for a confidence level of 95% with 20% margin of error [61]. From these experiments, we derived:

pTest — Mann-Whitney U test results which evaluates *pSRS* values are smaller than *pURS* values with 95% confidence [44]. "Pass" implies *pSRS* is smaller, and "Fail" otherwise.

Table 7 shows our results for each $\mathbb{NFM}$.

For all systems, we observed that SRS finds configurations with smaller build sizes with 95% confidence. Although the actual rank of configurations sampled is unknown, the results are consistent

$\mathbb{NFM}$	***pSRS***	***pURS***	***pTest***
axTLS	0.23	0.24	Pass
Fiasco	25.64	26.74	Pass
uClibc-ng	1.10	1.32	Pass

Table 7: Finding Near-Optimal Configs for KConfig Systems

with FSE2015 $\mathbb{NFM}$s as: 1) *rURS* does not depend on the size of the configuration space, but how many samples are collected, and 2) *pSRS*<*pURS*, which corresponds to *rSRS*<*rURS*.

These results show that SRS can perform accurate statistical reasoning over numerical features as well, while also showing that $\mathbb{BBPF}$ allows SRS to deal with numerical features without modifying the algorithm or the solver it uses. However, we believe that SRS can be enhanced to derive more accurate reasoning on numerical features, which may increase the *rPass* value. We leave this as a future work.

Conclusion: $\mathbb{BBPF}$ can be used by existing SAT analysis on SPLs "as is", with the work of [51] as an example.

Threats to Validity

Internal validity. To control randomness, we conducted 97 experiments and averaged the results for a confidence level of 95% with a 10% margin of error [61]. One exception is the result for KConfig ℕ𝔽𝕄s in **RQ4**, which repeated the experiments 25 times for 95% confidence and 20% margin of error [61], due to the lengthy time to build sampled configurations.

For **RQ2** and **RQ3**, we utilized the method for counting and URS configurations that are either proposed by the developers of the tool or practiced by prior work in SPL research.

For **RQ4**, we reduced the noise on the performance measurement of the samples as much as possible. FSE2015 ℕ𝔽𝕄s use the performance measurement from [32], which was used in prior works as well. KConfig ℕ𝔽𝕄s measured build sizes, as they are less susceptible to environmental influences.

External validity. We used 7 real-world systems with different numbers of features, number of clauses, and domains. Systems had different combinations of constraints with each other, so that we could evaluate our approach with different complexity of ℕ𝔽𝕄s. We are aware that our results may not generalize to all SPLs. At least, our results show identical trends across systems, which provides confidence that our conclusions should hold for many SPLs with comparable size of the configuration space.

We are also aware that Z3 and Clafer may not be representative of all CMT and CP solvers. At least, we used the tools that were widely used in SPL research, which are likely to be used in future SPL research as well.

5 RELATED WORK

Adding to Section 2, we discuss other relevant work here.

5.1 ℕ𝔽s in ℕ𝔽𝕄s

Most papers, for various reasons, did not describe how numerical variables were represented as features. Some considered ℕ𝔽s in the same manner as mandatory Boolean features, so that they had only one value [11, 38]. Some encoded ℕ𝔽s as alternative features, where each value of a ℕ𝔽 was considered a distinct feature [41]. Shi [57] used a single type of feature called 'pseudo-Boolean features'. In his work, Successor (+1) and Predecessor (-1) were introduced as a new type of constraint. As described in Section 3, representing alternative features as a propositional formula has limited scalability as the number of clauses grow rapidly as number of features increases.

Numerical variables and string-attributed feature models have been formalized. *Extended, Advanced or Attributed feature models* appear in the literature as a way to expand classical feature models. Attributed feature models extend Boolean feature models to include additional information about features [8, 10, 54]. In these works, the authors represent packages of attributes (*eg.,* cost, performance) bound to every Boolean feature in the extended feature model. Those attributes are not ℕ𝔽s [59]. The main differences between attributes and ℕ𝔽s are:

- Currently, there is no consensus on a notation to define attributes. However, most proposals agree that an attribute should consist at least of a name, a domain and a value [8], while a ℕ𝔽 consists of a name and a domain [40].
- A ℕ𝔽 is a feature, so it can be selected or deselected; it can have a value of zero, or it can have any value, and all these states are different. An attribute, in contrast, cannot be selected/un-selected [8].
- Every Boolean feature in an extended feature model is associated with a set of attributes [40]. A ℕ𝔽 in a ℕ𝔽𝕄 has a parent, and is affected by cardinality relationships [25].
- A set of attributes can contain several variables. Additionally, those variables can be present in different sets at the same time, as their respective value can be distributed along several sets belonging to different features [8]. Instead, Boolean and ℕ𝔽s are declared just once within the Feature Model [21].
- If we modify the value of just one ℕ𝔽 in a configuration, we are producing another configuration in the configuration space. That does not happen with attributes [40].

In any case, constraints are similarly formalized for both ℕ𝔽s and attributes [8].

5.2 Automated Reasoning of SPLs

As SPLs have many features and complex constraints, automated solvers were used to solve them as *Constraint Satisfaction Problems* (CSP). SAT, SMT, CP and *Binary Decision Diagrams* (BDDs) can be considered as different types of CSPs.

For classical feature models, SAT and BDD were the two most utilized automated solvers. For both, a feature model needs to be encoded as a propositional formula SAT solvers and BDDs were utilized in various analyses, including: checking if a feature model has conflicting constraints or deriving a valid configuration [7, 63], analyzing the structure of the FM [11, 31, 42], counting number of valid configurations [51, 63], and finding inconsistencies between code and feature model [41, 50]. SAT solvers and their variants, such as MaxSAT and #SAT solvers, were used to count the number of configurations and generate samples for testing and finding optimal configurations [18, 35, 38, 58].

For ℕ𝔽𝕄s, SMT and CP solvers were used as they natively support representation and reasoning of ℕ𝔽s and constraints. Encoding feature models for SMT and CP solvers is similar to that of a SAT solver except that they allow numerical variables and operations. Each variable represents a feature and constraints are represented with logical or arithmetic operations. As SMT and CP solvers have similar functionality as SAT solvers, they had similar usage in SPL research: finding conflicts between constraints [9, 21, 46], deriving valid configurations under user-imposed constraints [48], and generating samples for finding optimal configurations [34, 56, 58].

5.3 Solvers using Bit-Blasting

Bit-blasting can, computationally speaking, exhaust any solver if the input formula contains numerical values with large bit-width or complex arithmetic. Then, a pre-processing and simplification of the input formula is essential for reasoning efficiency.

In [20], the authors describe several classes of simplification methods implemented in the solver MathSAT_5, which are applied with certain heuristics like canonization (*eg.,* $X-X=0$), unconstrained propagation, packet splitting [5] and disjunctive partitioning [16] (*ie.,* the formula is increasingly processed in batches). Approaches like MathSAT_5 are elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations over bit-vectors; inequalities are not considered [15]. Albeit bit-vector theory admits quantifier elimination by considering that a fix-width is the maximum-width among all variables, this is rarely a practical approach. Instead, equisatisfiable formulas are used [39].

Solvers Z3 [22] and Yices [23] apply bit-blasting to every operation besides equality, which is, then, handled by a specialized solver. They also add axioms, dynamically, from array theory. Boolector [14] applies bit-blasting to bit-vector operations and lazily instantiates definitions of array axioms and macros.

A more recent solver is CVC4 [4]. It is a lazy and layered solver, which tries to decide satisfiability using faster, but incomplete, sub-solvers for inequality reasoning. In case of sub-solvers are not enough, theory lemmas and propagated literals are added to the formula, and a lazy CNF-SAT bit-blasting solver is employed. STP [30] performs several array optimizations, as well as arithmetic and Boolean simplifications on the bit-vector formula before bit-blasting to MiniSat [60].

5.4 Uniform Sampling of SPLs

URS is not simple, as merely random selecting features rarely yield valid configurations [43]. Chakraborty et al. [17] proposed Unigen2, a uniform sampling algorithm for propositional formula based on an approximate #SAT solver. Dutra et al. [24] proposed QuickSampler, a sampling algorithm for efficiently generating valid configurations for testing. On these algorithms, Plazar et al. [52] showed that Unigen2 is not scalable for configuration spaces larger than 10^{10}, which is not applicable for our Kconfig $\mathbb{NFM}$s, while QuickSampler samples are often not uniformly distributed.

We used Smarch [38], a URS algorithm that can scale up to configuration spaces of size 10^{249}.

5.5 Statistical Analyses of SPLs

Prior work on SPLs performed statistical analyses to reason on colossal ($\gg 10^{82}$) and complex configuration spaces. To estimate the influence of a feature on performance, samples were benchmarked and compared for performance differences [33, 55]. To find optimal configurations, samples were used to search the configurations throughout the space [18, 26, 34, 35, 51, 56]. To evaluate different sampling approaches to locate variability bugs, URS was considered to be the baseline to compare with other approaches [1, 47, 62].

6 CONCLUSIONS AND FUTURE WORK

Configuration spaces grow exponentially with increasing number of features, which makes statistical reasoning crucial for understanding them. Compared to classical feature models, $\mathbb{NFM}$s have comparatively larger and more complex configuration spaces due to increased variability and additional types of constraints. This makes statistical reasoning of $\mathbb{NFM}$s even more vital. Well-known automated solvers that handle numerical variables, however, were not feasible for counting and URS of configurations for $\mathbb{NFM}$s, which are needed for unbiased statistical reasoning of product spaces.

We evaluated bit blasting to encode $\mathbb{NF}$s and their constraints as propositional formulas, to utilize existing SAT-based approaches on counting and URS configurations. With bit blasting, $\mathbb{NFM}$s were represented as binary bits while their constraints were represented as propositional clauses.

Our experiments showed bit-blasting:

- can represent $\mathbb{NF}$s and their constraints up to 10 bits of accuracy without overhead,
- can utilize sharpSAT to count the number of configurations, which was much faster and more scalable than current SMT and CP solvers,
- can utilize Smarch [38], an existing tool to URS configurations, while SMT and CP could not guarantee the uniform distribution of their produced samples, and
- was able to use SRS [51], a previously published algorithm, to find near-optimal configurations for classical feature models, to search for near-optimal configurations in $\mathbb{NFM}$s as well, and
- the largest KConfig $\mathbb{NFM}$s that we examined had a huge configuration space 10^{45} (see Table 2); we believe much larger configuration spaces can be analyzed.

We are confident our work can be utilized by others to analyze different SPLs with $\mathbb{NF}$s. Our research also suggests future explorations:

- expand bit-blasting to handle more arithmetic operations,
- evaluate whether other prior work on analyzing classical feature models with SAT solvers can be extended by bit-blasting.

ACKNOWLEDGMENTS

Work by Munoz, Pinto and Fuentes is supported by the projects MAGIC P12-TIC1814, TASOVA MCIU-AEI TIN2017-90644-REDT, HADAS TIN2015-64841-R and MEDEA RTI2018-099213-B-I00 (last two co-financed by FEDER funds). Work by Oh and Batory is supported by NSF grants CCF-1212683 and ACI-1550493. This work has been supported by IMFAHE Foundation-Fellowship of 2018

REFERENCES

[1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. In *ACM SIGPLAN Notices*, Vol. 52. ACM, ACM, ACM, 144–155.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines.* Springer, NY, USA.

[3] Clark Barrett. 2013. Decision Procedures: An Algorithmic Point of View, by Daniel Kroening and Ofer Strichman, Springer-Verlag, 2008. *Journal of Automated Reasoning* 51, 4 (2013), 453–456.

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, Springer, NY, USA, 171–177.

[5] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Splitting on demand in SAT modulo theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, Springer, NY, USA, 512–526.

[6] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, NY, USA, 305–343.

[7] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, Springer, NY, USA, 7–20.

[8] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[9] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Java CSP solvers in the automated analyses of feature models. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, Springer, NY, USA, 399–408.

[10] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2006. A first step towards a framework for the automated analysis of feature models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms* 0 (2006), 39–47.
[11] Thorsten Berger, Steven She, Rafael 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.
[12] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press, IEEE.
[13] Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *2009 Formal Methods in Computer-Aided Design*. IEEE, IEEE, Piscataway, NJ, USA, 69–76.
[14] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 174–177.
[15] Randal E Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A Seshia, Ofer Strichman, and Bryan Brady. 2007. Deciding bit-vector arithmetic with abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 358–372.
[16] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. 1997. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proceedings of the 34th annual Design Automation Conference*. ACM, ACM, New York, NY, USA, 728–733.
[17] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319.
[18] Jianfeng Chen, Vivek Nair, Rahul Krishna, and Tim Menzies. 2018. " Sampling" as a Baseline Optimizer for Search-based Software Engineering. *IEEE Transactions on Software Engineering* 0 (2018), 12.
[19] Maciej Ciesielski, Walter Brown, and André Rossi. 2013. Arithmetic bit-level verification using network flow model. In *Haifa Verification Conference*. Springer, Springer, NY, USA, 327–343.
[20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, Springer, 93–107.
[21] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, IEEE, IEEE, 472–481.
[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, NY, USA, 337–340.
[23] Bruno Dutertre. 2014. Yices 2.2. In *International Conference on Computer Aided Verification*. Springer, Springer, NY, USA, 737–744.
[24] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, ACM, New York, NY, USA, 549–559.
[25] Holger Eichelberger and Klaus Schmid. 2013. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*. ACM, ACM, New York, NY, USA, 12–21.
[26] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An empirical study of configuration mismatches in linux. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*. ACM, ACM, New York, NY, USA, 19–28.
[27] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. 2010. Encoding industrial hardware verification problems into effectively propositional logic. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, Springer, NY, USA, 137–144.
[28] The Linux Foundation. 2018. Kconfig language specification. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.
[29] The Linux Foundation. 2018. Kconfig tool specification. https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt.
[30] Vijay Ganesh and David L Dill. 2006. System description of STP.
[31] Paul Gazzillo. 2017. Kmax: finding all configurations of Kbuild makefiles statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, ACM, ACM, 279–290.
[32] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. 2014. Experiments on optimizing the performance of stencil codes with spl conqueror. *Parallel Processing Letters* 24, 03 (2014), 1441001.
[33] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 301–311. https://doi.org/10.1109/ASE.2013.6693089
[34] Jianmei Guo, Jia Hui Liang, Kai Shi, Dingyu Yang, Jingsong Zhang, Krzysztof Czarnecki, Vijay Ganesh, and Huiqun Yu. 2017. SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. *Software & Systems Modeling* 0 (2017), 1–20.
[35] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. 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, IEEE/ACM, Piscataway, NJ, USA, 517–528.
[36] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. 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 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 517–528. http://dl.acm.org/citation.cfm?id=2818754.2818819
[37] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2018. Variability models for generating efficient configurations of functional quality attributes. *Information and Software Technology* 95 (2018), 147–164.
[38] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling from Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.
[39] Martin Jonas. 2016. *SMT Solving for the Theory of Bit-Vectors*. Ph.D. Dissertation. Masaryk University.
[40] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. 2013. From extended feature models to constraint logic programming. *Science of Computer Programming* 78, 12 (2013), 2295–2312.
[41] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, IEEE/ACM, Piscataway, NJ, USA, 805–824.
[42] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, IEEE/ACM, Piscataway, NJ, USA, 91–100.
[43] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, ACM, New York, NY, USA, 81–91.
[44] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* 0, 0 (1947), 50–60.
[45] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.
[46] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*. ACM, IEEE/ACM, Piscataway, NJ, USA, 18–21.
[47] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, IEEE/ACM, Piscataway, NJ, USA, 643–654.
[48] Daniel-Jesus Munoz, José A. Montenegro, Mónica Pinto, and Lidia Fuentes. 2019. Energy-aware environments for the development of green applications for cyber-physical systems. *Future Generation Computer Systems* 91 (2019), 536 – 554. https://doi.org/10.1016/j.future.2018.09.006
[49] Daniel-Jesus Munoz, Mónica Pinto, and Lidia Fuentes. 2018. Finding correlations of features affecting energy consumption and performance of web servers using the HADAS eco-assistant. *Computing* 100, 11 (01 Nov 2018), 1155–1173. https://doi.org/10.1007/s00607-018-0632-7
[50] Sarah Nadi and Ric Holt. 2012. Mining Kbuild to detect variability anomalies in Linux. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, IEEE/ACM, Piscataway, NJ, USA, 107–116.
[51] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, IEEE/ACM, Piscataway, NJ, USA, 61–71.
[52] 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 *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*. HAL-Inria, Xian, China, 1–12. https://hal.inria.fr/hal-01991857
[53] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier, Elsevier.
[54] Luis Emiliano Sánchez, Jorge Andrés Diaz-Pace, and Alejandro Zunino. 2019. A family of heuristic search algorithms for feature model optimization. *Science of Computer Programming* 172 (2019), 264 – 293.

[55] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE/ACM, Piscataway, NJ, USA, 342–352.

[56] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. 2013. 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/ACM, ASE, 465–474. https://doi.org/10.1109/ASE.2013.6693104

[57] K. Shi. 2017. Combining Evolutionary Algorithms with Constraint Solving for Configuration Optimization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE/ACM, Piscataway, NJ, USA, 665–669. https://doi.org/10.1109/ICSME.2017.32

[58] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 284–294. https://doi.org/10.1145/2786805.2786845

[59] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-interaction Detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 167–177. http://dl.acm.org/citation.cfm?id=2337223.2337243

[60] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.

[61] Creative Research Systems. 2019. Sample Size Calculator. https://www.surveysystem.com/sscalc.htm.

[62] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration Coverage in the Analysis of Large-scale System Software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM, New York, NY, USA, Article 2, 5 pages. https://doi.org/10.1145/2039239.2039242

[63] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[64] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 424–429.

[65] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. https://doi.org/10.1007/978-3-642-81955-1_28

[66] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. 2013. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design* 42, 1 (2013), 3–23.

A ARTIFACT INFORMATION

The artifact for this paper contains a *Virtual Machine* (VM) with pre-built and configured tools to re-run the evaluation, including: Bit-blasting for ℕ𝔽s, model-counting, URS, and SRS for three different types of solvers. A VM is provided due to the amount of knowledge and time necessary to configure and run the third-party and new tools. A Linux operating system is mandatory to run those tools, while a VM can run on almost any operating system and/or hardware. It also includes the tools that natively supports ℕ𝔽s - Clafer and Z3py. Tested feature models and their intermediate and final results are also included.

A.1 Access and Content

A VM is pre-configured to re-create the experiments, as well as to reuse for different ℕ𝔽𝕄s and/or data-sets. The VM and its detailed instructions are available at:

https://github.com/danieljmg/SPLs-BitBlasting-URS

The VM make use of the following **third-party** assets:

- Lubuntu 18.04 LTS x86_64 operating system [12].
- The Python Interpreter version 3.7 [13].
- The Oracle's open-source Java Development Kit [14].
- Clafer Instance Generator 0.45 [15].
- The Z3 theorem prover SMT solver for Python (Z3py) [16].
- Model counting SAT solver (SharpSAT) [17].
- JetBrains 2019.1 Python IDE (PyCharm) [18].
- The Smarch random sampling tool [38].
- The Kolmogorov-Smirnov Test (KS-t) [19].
- The Mann-Whitney U Test (MWU-t) [20].
- The Oracle VirtualBox virtualization software [21].
- A KConfig measurement VM [22].

The VM includes the following **new** assets:

- Clafer, Z3py and DIMACS (SharpSAT format) ℕ𝔽𝕄s of the seven SPLs in Table 2.
- A Python script to transform numerical features modeled in Z3py into Tseitin-CNF DIMACS format using Bit Blasting. It supports composed first order logic and linear arithmetic with integers as in Table 1.
- Scripts to count the number of configurations from Clafer, Z3py and DIMACS models.
- Scripts to random sample configurations from Clafer, Z3py and DIMACS models.
- A Python script to rank sets of random samples to evaluate their uniform distribution. A set of samples is obtained and measured from different reasoners.
- Intermediate files including sets of samples, ranks, and measurements.

A.2 Installation and Environment Overview

Running the evaluation has following minimum requirements:

- A machine with at least 4GB of memory RAM and 10GB of disk free space, with x86 64-bit operating system and Oracle VirtualBox 6 installed.[23]
- Intel VT-x or AMD-V CPU option activated in the motherboard BIOS settings. However, RQ4 is partially not compatible with Intel VT-x.

To set up the environment, you first need to load the downloaded VM into VirtualBox clicking *File->Import Appliance* and searching for *SPLC19VM.ova*. Lubuntu credentials are:

- **User**: caosd
- **Password** and **Sudoers password**: splc19

After Lubuntu is ready to use, in its *Desktop* we can find:

- Folder *featuremodels* where all ℕ𝔽𝕄s with different formats (Clafer, DIMACS and Z3py) are located.
- Folder *samples* where 100, 300 and 500 pre-computed samples for each solver and ℕ𝔽𝕄s are located.
- Folder *UFscripts* where scripts for model count and sampling with SharpSAT are located.
- Launcher for PyCharm Python IDE.
- Launcher for LXTerminal in order to execute scripts.

A.3 Usage Summary

Different scripts are provided for each research question (RQ):

- **RQ1**: Open *PyCharm* and run *Z3toCNF.py* to Bit-blast ($a+b > c$), and finish with *UFscripts/SharpSAT_ABGTC* scripts.
- **RQ2**: Run the scripts at *UFscripts/SharpSATCounting*, *UFscripts/ClaferCounting* and *PyCharm/Z3*.
- **RQ3**: Run the scripts at *UFscripts/ClaferSampling*, *PyCharm/Z3*, *PyCharm/ranksampling*, *HCS_Optimizer/randtest.py* and *HCS_Optimizer/evaluation.py*. Finish with the KS-Test.
- **RQ4**: Adjust and run */search.py* and, for Kconfig models, use the *KConfig measurement VM* ending with the *MWU-test*.

Adjustments required for each RQ is indicated in the code. Data comparisons and graphs can be performed with the included software *Gnumeric* [24]. Detailed steps can be found in the *Github* artifact's page.

[12] https://lubuntu.net/
[13] https://www.python.org/
[14] https://openjdk.java.net/
[15] https://github.com/gsdlab/claferIG
[16] https://github.com/Z3Prover/z3
[17] https://github.com/marcthurley/sharpSAT
[18] https://www.jetbrains.com/pycharm
[19] https://www.wessa.net/rwasp_Reddy-Moores%20K-S%20Test.wasp
[20] https://www.socscistatistics.com/tests/mannwhitney/default2.aspx
[21] https://www.virtualbox.org/
[22] https://github.com/paulgazz/kconfig_case_studies
[23] https://www.virtualbox.org/wiki/Downloads
[24] http://www.gnumeric.org/