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

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

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

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

© 2019 Association for Computing Machinery.

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 (NFs). feature models with NFs are Numerical Feature Models (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 (e.g., [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 NFMs, 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, 18].

Bit-blasting encodes numerical values as binary bits and represents 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 SAT 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 #NFs as a Bit-Blasted Propositional Formula (BBPF).

BBPF can be input to existing #SAT-based tools for counting and URS solutions of a #NF, which SMT and CP solvers cannot do. In this way #NFs 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 #NF into a #BBPF,
- Experiments that show counting and URS solutions of #BBPF outperform SMT and CP solvers, and
- Evaluation of known SPL analyses using #NFs with huge configuration spaces, the largest exceeding $10^{19}$ 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 ($\neq$, $>$, $\geq$), 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 #NFs that contain both binary features and #NFs [36]. An #NF has a name $N$, a type (ie., domain), and range ($e.g., N \in [1, 2, ... 128]$). #NFs add new constraints to the set of propositional connectives, including: numerical equality (=), numerical inequalities ($\neq$, $>$, $\geq$ and $\leq$), 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 #NFs 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 ($e.g., AES\textunderscore crypto \Rightarrow key\textunderscore size > 128$) 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 $\phi$ be the propositional formula of a classical feature model. Let $S(\phi)$ be the set of all solutions of $\phi$. Each solution of $\phi$ 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 $\phi$. #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 in a fixed linear ordering of $S(\phi)$. By construction, URS of numbers in $[1..|S(\phi)|]$ is isomorphic to URS of configurations in $S(\phi)$.

Oh et al. [51] showed that $c_{best}$ will be, on average, within the top $1\%$ 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 #NFs

We describe how to integrate bit-blasting and classical feature models to form #NFs and how to translate a #NF 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) or signed (binary two’s complement encoding). We use the Little-Endian representation, i.e., signed bit-vectors, where the last bit encodes the sign as positive (0) or negative (1).

Table 1 shows examples of two’s complement 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] (i.e., n = 3 bits) where a3 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 (≠) is a bit-by-bit disjunction of logical XORs (∨) (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 ≥ (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 C0, 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 satisiability and number of configurations.

3.2 Producing a BBPF for an NFM

We encode the Boolean features and their constraints of an NFM as a propositional formula in the standard way [2]. Then, NFMs and their constraints of a NFM are encoded as propositional clauses making use of the Z3 solver ‘Tactic’ functionality. We conjoin both predicates (or substitute them below) to form the BBPF for that NFM. Here are some details:

**NF Definition.** Let a signed NF f have range [a, b]. Bit-blasting uses \(\log_2(\max(|a|, |b|)) + 1\) variables to represent the bits of f, where 1 variable encodes the sign. Propositional clauses for two constraints (f ≥ a) and (f ≤ 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 \(\log_2(b − a) + 1\).

We represent all NFs 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.

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

Two NFs bounded under the same constraint may have different bit-widths due to different value ranges. As the bit-width of each NF is fixed, the NF with shorter bit-width needs to be extended to

---

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.
match the bit-width of the other NFM. Extending the bit-width does not change an NFM’s possible values due to range constraints.

Mixed Boolean and NFM Constraints. A numerical constraint can be qualified by Boolean features, such as \( a = b \neq 0 \), where \( a \) is a Boolean feature and \( b \) is an NFM. In this case, the propositional clauses for NFM operations can be generated first (e.g., let \( \omega \) be the bit-blasted propositional formula of \( (b \neq 0) \)), which is then substituted into the original formula to yield the result, namely \( a = \omega \).

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

Alternative Features. For a large set of alternative features, representing them as an NFM 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 NFM requires only \( n \) bits. Regarding the clauses, alternative features requires \((\sum_0^n) + 1\) CNF clauses, while an NFM requires none. A NFM that allows multiple discrete values (e.g., odd numbers \([2, 3, 7, 11, 13, \ldots]\)) instead of values within a range can be encoded in the same manner.

Fig. 1 shows our encoding of an NFM as a BBFF. 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 NFM is conjoined at lines 4–6. As the NFM ‘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 NFM, 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 NFM. We answer the following research questions to evaluate BBFF:

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

Figure 1: An example of NFM to BBFF Conversion

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

We used real-world NFM s 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 NFM s and/or difference configuration space size. Henceforth, we use FSE2015 to denote the feature models from Siegmund et al. [58].

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

Compared to FSE2015 NFM s, KConfig models have many more features and have huge configuration spaces. The KConftool [58] derives a propositional formula for each KConftool. As KClasse simplifies NFM s to have their default values only, we augmented their formula with bit blasting to allow different NFM values.

The KConftool NFM s that we examined had NFM s as small as \([0, 1]\) and as large as \([0, 2^{32} - 1]\). When the range of a NFM is not defined in a KConftool model, any value within the range of the integer data type is possible, which is \([0, 2^{32} - 1]\). For NFM s 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 NFM and constraints for BBFF, we used the formula printing functionality of the Z3 solver. To count the number of configurations in BBFF, we used sharpSAT [64], a state-of-art model counter for propositional formulas. To sample configurations of BBFF, we used Smarch [38], a state-of-art tool for URS propositional formula solutions.

**RQ1**: How many bits per NFM 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 NFM s of unsigned integers. We consider this constraint as an upper bound on the overhead of numerical constraints.\(^5\) Propositional formulas with three \(b\)-bit NFM s related by this constraint were generated and

\(^{5}\) Actual numerical constraints were simpler, as \(C\) was substituted with a constant. Constants in constraints simplifies the formula by Boolean constraint propagation.
benchmark the number of bits for counting configurations.

For each formula, we measured:

- \( NC \) = 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 \( NC \), \( tC \), 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:

- \( NC \) grew linearly with increasing \( #b \),
- \( tC \) grew exponentially with increasing \( #b \), and
- \( tS \) was below 1 second for \( #b \leq 13 \).

The linear increase of \( NC \) is due to the use of Tseitin’s transformation in generating CNF formulas. As the number of \( NF \) variables increases with linearly with \( #b \), Tseitin’s transformation guarantees a linear increase of \( O(3^n) \) 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 \( NFs \) with constraints. The fact that there was a time-out after 16 bits does not mean that \( NFs \) larger than 16 bits cannot be treated by bit-blasting. When a \( NF \) 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 \( NF \) of range \([0, 2^{12} – 1]\) can be discretized into a 10-bit \( NF \) with the precision of \( 2^{12} \). This makes analyses feasible by reducing the precision of possible values.

Conclusions: 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 \( BBPF \) 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 Clafner with the Choco solver\(^7\) as a representative CP solver. Z3 and Clafner 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.\(^8\)
- Clafner has an internal functionality to count configurations, by using the option ‘--t’. Its functionality involves enumerating configurations as well.\(^9\)


\( ^7 \) Clafner, https://www.clafner.org/.


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% 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 NFM s 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 a given configuration is easy. In KConfig systems, we do not know the exact rank of the samples. 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 show a similar trend with the FSE2015 systems. Even for models with larger range NFM s and larger configuration spaces, Smarch was able to sample configurations within a reasonable time. Smarch passes KS tests for all NFM s and sample sizes, which says that Smarch performs URS with 95% confidence, and Z3 and Clafer failed KS tests for some NFM s 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 NFM s. Note that, we could not check the randomness of the samples in the same manner a FSE2015 NFM s, 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 show a similar trend with the FSE2015 systems. Even for models with larger range NFM s and larger configuration spaces, Smarch was able to sample configurations within a reasonable time. Smarch passes KS tests for all NFM s 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_\text{best} – the best performing configuration, would be in the top \( \frac{1}{(n+1)} \) percentile of all

<table>
<thead>
<tr>
<th>Type</th>
<th>Model</th>
<th>BBPF</th>
<th>Z3</th>
<th>Clafer</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSE2015</td>
<td>Dune</td>
<td>26.20s</td>
<td>11s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>HSMGP</td>
<td>40.70s</td>
<td>14s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>HiPAcc</td>
<td>458s</td>
<td>33s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>Trimesh</td>
<td>Time-out</td>
<td>2s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>axTLS</td>
<td>Time-out</td>
<td>Time-out</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>Fiasco</td>
<td>Time-out</td>
<td>Time-out</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>uclibc-ng</td>
<td>Time-out</td>
<td>Time-out</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

**Table 3: Average Counting Time Comparison**

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 sec,
- Smarch passed KS tests for all NFM s and sample sizes, which says that Smarch performs URS with 95% confidence, and
- Z3 and Clafer failed KS tests for some NFM s and sample sizes. This says that the randomization options for Z3 and Clafer do not always achieve URS.

---

9We 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.
 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.

Their work, however, focused on feature models with Boolean features and constraints, while optimizing feature models with NFM and constraints was left as future work. We wanted to see if their work could be used as is with EBFT.

We replicated SRS to find near-optimal configurations from the NFM's of RQ2. For FSE2015 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 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 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 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 NFM's, as the number of features, constraints, and how a feature affects the objective to optimize are different for each NFM.
- \( rSRS \) was lower than \( rURS \) for all NFM's with 95% confidence, which indicates SRS outperforms URS.
- \( pSRS \) was lower than \( pURS \) for all 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 NFM's, sorted by their performance. The X-axis denotes

11 We used Mann-Whitney U Test as the distributions of the results are non-parametric.
Table 6: Finding Near-Optimal Configurations for FSE2015 Systems

<table>
<thead>
<tr>
<th>NFM</th>
<th>N</th>
<th>rSRS</th>
<th>rURS</th>
<th>rTest</th>
<th>pSRS</th>
<th>pURS</th>
<th>pTest</th>
<th>rBetter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dune</td>
<td>71.32</td>
<td>0.007</td>
<td>0.016</td>
<td>Pass</td>
<td>0.039</td>
<td>0.042</td>
<td>Pass</td>
<td>93%</td>
</tr>
<tr>
<td>HSMGP</td>
<td>66.42</td>
<td>0.008</td>
<td>0.017</td>
<td>Pass</td>
<td>0.005</td>
<td>0.011</td>
<td>Pass</td>
<td>91%</td>
</tr>
<tr>
<td>HiPAcc</td>
<td>65.82</td>
<td>0.010</td>
<td>0.017</td>
<td>Pass</td>
<td>0.002</td>
<td>0.004</td>
<td>Pass</td>
<td>82%</td>
</tr>
<tr>
<td>Trimesh</td>
<td>129.21</td>
<td>0.003</td>
<td>0.009</td>
<td>Pass</td>
<td>0.003</td>
<td>0.013</td>
<td>Pass</td>
<td>91%</td>
</tr>
</tbody>
</table>

Figure 3: Configuration Space of FSE2015 Systems

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:

\[ p_{Test} = \text{Mann-Whitney U test results which evaluates } p_{SRS} \text{ values are smaller than } p_{URS} \text{ values with 95\% confidence [44].} \]

"Pass" implies \( p_{SRS} \) is smaller, and "Fail" otherwise.

Table 7 shows our results for each NFM.

Table 7: Finding Near-Optimal Configs for KConfig Systems

<table>
<thead>
<tr>
<th>NFM</th>
<th>pSRS</th>
<th>pURS</th>
<th>pTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>axTLS</td>
<td>0.23</td>
<td>0.24</td>
<td>Pass</td>
</tr>
<tr>
<td>Fiasco</td>
<td>25.64</td>
<td>26.74</td>
<td>Pass</td>
</tr>
<tr>
<td>uClibc-ng</td>
<td>1.10</td>
<td>1.32</td>
<td>Pass</td>
</tr>
</tbody>
</table>

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 with FSE2015 NFM s as: 1) rURS does not depend on the size of the configuration space, but how many samples are collected, and 2) \( p_{SRS} < p_{URS} \), which corresponds to \( r_{SRS} < r_{URS} \).

These results show that SRS can perform accurate statistical reasoning over numerical features as well, while also showing that 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 $r$Pass value. We leave this as a future work.

**Conclusion:** BBPT 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 NFs 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 NFs 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 (e.g., 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 NFMs 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 among 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 are 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].

Nevertheless, 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. Boolec- tor [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 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 $10^{65}$. 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].

5.5 Statistical Analyses of SPLs

Prior work on SPLs performed statistical analyses to reason on colossal ($\gg 10^{52}$) 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, NFM’s have comparatively larger and more complex configuration spaces due to increased variability and additional types of constraints. This makes statistical reasoning of NFM’s even more vital. Well-known automated solvers that handle numerical variables, however, were not feasible for counting and URS of configurations for NFM’s, which are needed for unbiased statistical reasoning of product spaces.

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

Our experiments showed bit-blasting:

- can represent NFM’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 NFM’s as well, and
- the largest Kconfig NFM’s that we examined had a huge configuration space $10^{65}$ (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 NFM’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


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 NFM’s, 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 support NFM’s - 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 NFM’s 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.
- The Python Interpreter version 3.7.
- The Oracle’s open-source Java Development Kit.
- Clafer Instance Generator 0.45.
- The Z3 theorem prover SMT solver for Python (Z3py).
- Model counting SAT solver (SharpSAT).
- JetBrands 2019.1 Python IDE (PyCharm).
- The Kolmogorov-Smirnov Test (KS-t).
- The Smarch random sampling tool [38].
- The Mann-Whitney U Test (MWU-t).
- 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.

The VM includes the following new assets:

- Clafer, Z3py and DIMACS (SharpSAT format) NFM’s 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.
- 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 SPLIC19VM.ovf. Lubuntu credentials are:

- User: caosd
- Password and Sudoers password: splet19

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

- Folder featuremodels where all NFM’s with different formats (Clafer, DIMACS and Z3py) are located.
- Folder samples where 100, 300 and 500 pre-computed samples for each solver and NFM’s 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/Randomtest.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. Detailed steps can be found in the Github artifact’s page.

---

23https://lubuntu.net/
24https://www.python.org/
25https://openjdk.java.net/
26https://github.com/paulgazz/kconfig_case_studies
27https://github.com/marcthurley/sharpSAT
29https://www.virtualbox.org/
30http://www.gnumeric.org/