# ℕ𝕖𝕞𝕠: A Tool to Transform Feature Models with Numerical Features and Arithmetic Constraints

Daniel-Jesus Munoz[0000-0002-1398-9423][1,2], Jeho Oh[3],
Monica Pinto[0000-0002-5376-742X][1,2], Lidia Fuentes[0000-0002-5677-7156][1,2]
, and Don Batory[3]

[1] ITIS Software, Universidad de Málaga, Spain
[2] CAOSD, Departamento LCC, Universidad de Málaga, Andalucía Tech, Spain
{danimg, pinto, lff}@lcc.uma.es
[3] Department of Computer Science, University of Texas at Austin, Texas, USA
{jeho, batory}@cs.utexas.edu

**Abstract.** Real-world *Software Product Lines* (SPLs) need *Numerical Feature Models* (ℕ𝔽𝕄s) whose features not only have boolean values satisfying boolean constraints, but also have numeric attributes satisfying arithmetic constraints. A key operation on ℕ𝔽𝕄s finds near-optimal performing products, which requires counting the number of SPL products. Typical constraint satisfaction solvers perform poorly on counting.

ℕ𝕖𝕞𝕠 (Numbers, features, models) supports ℕ𝔽𝕄s by *bit-blasting*, the technique that encodes arithmetic as boolean clauses. ℕ𝕖𝕞𝕠 translates ℕ𝔽𝕄s to propositional formulas whose products can be counted efficiently by #SAT solvers, enabling near-optimal products to be found. We evaluate ℕ𝕖𝕞𝕠 with a diverse set of real-world ℕ𝔽𝕄s, complex arithmetic constraints, and counting experiments in this paper.

**Keywords:** feature model · bit-blasting · propositional formula · numerical features · model counting · software product lines

## 1 Introduction

*Software Product Line* (SPL) engineering is a key reuse approach to build highly-configurable systems [1]. An SPL reduces the overall engineering effort to produce similar products by capitalizing on their commonalities and managing their configurations. A classical *Feature Model* (𝔽𝕄) defines SPL variability by boolean-valued features and boolean constraints, called *propositional formulas* (ℙ𝔽s). A ℙ𝔽 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* [2,5].

Real-world SPLs need *Numerical Feature Models* (ℕ𝔽𝕄s). An example is the SPL of Linux repositories where packages have versioning and other numerical attributes [29] called *Numerical Features* (ℕ𝔽s). Relationships among ℕ𝔽s are arithmetic constraints. In effect, ℕ𝔽𝕄s are 𝔽𝕄s with ℕ𝔽s.

SAT solvers find configurations of 𝔽𝕄s, because 𝔽𝕄s can be translated to ℙ𝔽s, and SAT is efficient for finding ℙ𝔽 solutions (ie., configurations). Unfortunately, SAT performs poorly on counting as it enumerates products, which is infeasible for large SPL product spaces, $\gg 10^6$ products [30].

Why is counting important? Because counting products enables unbiased statistical inferences on large product spaces [21,28]. That, in turn, can be used to find the best performing configuration in a user-constrained SPL product space given a defined workload [28,39].

Only a handful of automated solvers support ℕ𝔽𝕄s, namely *Satisfiability Modulo Theories* (SMT) [4] and *Constraint Programming* (CP) [34] solvers. Unfortunately, SMT and CP solvers cannot count and instead perform brute-force enumeration. In contrast, *#SAT* solvers extend SAT solvers to count the number of solutions of a ℙ𝔽 efficiently without enumeration [9]. #SAT solvers outperform SMT and CP solvers on counting. We use techniques to translate ℕ𝔽𝕄s into ℙ𝔽s [26]. Concretely, *bit-blasting* [11] encodes numerical values into bits and arithmetic constraints into ℙ𝔽s.

In this paper, we present ℕ𝕖𝕞𝕠 (Numbers, features, models) which natively supports ℕ𝔽𝕄s and efficient SAT operations to find ℕ𝔽𝕄 products (satisfying boolean and/or arithmetic constraints) as well as #SAT counting ℕ𝔽𝕄 products. ℕ𝕖𝕞𝕠's ℕ𝔽𝕄 language is simple; it supports constant, enumerated, and range variables, along with boolean and arithmetic constraints. Given an ℕ𝔽𝕄, ℕ𝕖𝕞𝕠 generates a ℙ𝔽 in the standard format for SAT-based tools. At which point, a SAT or #SAT tool can be invoked.

The novel contributions of our paper are:

- Explaining how ℕ𝕖𝕞𝕠 automatically translates and optimizes the encoding of arithmetic operations (as complex as multiplication, division, and modulo) and arithmetic constraints on ℕ𝔽s into ℙ𝔽s; and
- Experimentally comparing the run-time of ℕ𝕖𝕞𝕠 with popular SMT and CP solvers on processing bit-blasted ℙ𝔽s on artificial ℕ𝔽𝕄s and 7 real-world ℕ𝔽𝕄s with up to $10^{45}$ configurations using (1) benchmarks for arithmetic expressions and (2) benchmarks for counting tasks.

ℕ𝕖𝕞𝕠 is open-source: `https://github.com/danieljmg/Nemo_tool`

## 2 Background

### 2.1 Propositional Formulas of Feature Models

A classical feature model uses only boolean features but this very restriction allows it to be transformed into a ℙ𝔽, where features are boolean variables and constraints are clauses [2,5]. State-of-the-art tools that convert feature models into ℙ𝔽s are `FeatureIDE` [41] and `Glencoe` [35]; both translate a graphically-drawn feature model into a ℙ𝔽 in a *Conjunctive Normal Form* (CNF).

However, real-world SPLs use ℕ𝔽𝕄s that contain both binary features and ℕ𝔽s [17]. An ℕ𝔽 has a name N, a type (ie., domain), and range (eg., $N \in [1, 2, ...128]$).

ℕ𝔽𝕄s add arithmetic constraints to the set of propositional connectives. And arithmetic constraints can constrain boolean features and vice-versa.

Two examples of ℕ𝔽𝕄s are: (1) the HADAS eco-assistant [27] where energy parameters are represented as ℕ𝔽s in an integer domain, and propositional connectives and inequalities are present in cross-tree constraints (eg., `AEScrypto` ⇒ `keySize`>`128`) and (2) WeaFQAs [18] has integer and float attributes with propositional connectives and interval constraints (ie., numerical value ranges).

### 2.2 Bit-Blasting

*Bit-blasting*, also called *flattening*, is the transformation of a bit-vector arithmetic formula to a ℙ𝔽 [3]. Variables are bit-vectors and arithmetic operations are propositional clauses that reference bits. The resulting ℙ𝔽 is satisfiable whenever the original formula is. Our work focuses on basic arithmetic relations and operations, and of course, counting. We present operations in order of their usage frequency in real-world ℕ𝔽𝕄s [26]: equality (=), inequalities (≠, >, ≥), addition (+), subtraction (-), multiplication (*), division (/), and modulo (%).

## 3 Bit-Blasting Basic Arithmetic Operations

The main property of bit-vectors is their width which defines: a) the minimum and maximum limits of the original numerical variables, and b) if the vector is unsigned (ie., binary *sign-magnitude* encoding) or signed (ie., binary *two's complement* encoding).[4] We use the Big-Endian representation[5] where the first bit of the bit-vector encodes the sign as positive (0) or negative (1).

Table 1 shows examples of two's complement bit-blasting ℙ𝔽s for arithmetic relations on Big-Endian signed integers with a value range of [-4,3] (ie., $\mathtt{n} = 3$ bits) where bit-`1` is the integer sign:

$$\mathtt{a}, \mathtt{b} = < \mathtt{a}_1, \mathtt{a}_2, \ldots \mathtt{a}_\mathtt{n} >, < \mathtt{b}_1, \mathtt{b}_2, \ldots \mathtt{b}_\mathtt{n} > \quad \bigwedge \quad \text{where } \mathtt{a}_\mathtt{i}, \mathtt{b}_\mathtt{i} \in \{0, 1\}; 1 \leq \mathtt{i} \leq \mathtt{n}$$

Of course, we could have used larger widths in Table 1, but $\mathtt{n} = 3$ is sufficient to grasp the encoding patterns. Equality (==) is the conjunction of bitwise equivalences (row 1, col ℙ𝔽). Inequality (≠) is a bit-by-bit disjunction of `XORs` (⊕) (row 2, col ℙ𝔽). After the numerical sign comparison (first clause of col PF in rows 3 and 4), there are bit-by-bit equivalences until 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).

Arithmetic encoding patterns are more complex. Addition and multiplication of bit-vectors can produce a result outside the domain range. For example, for `3`

[4] Two's complement negative integer encoding is the binary complement of the positive encoding plus one bit.

[5] Big-Endian: An order of bits in which the 'Big end' (most significant value in the sequence) is first in the sequence.

**Table 1.** Propositional Formulas for 3-bit Two's Complement Signed Integers

| Row | Operation | Bit-Blasted Model | Propositional Formula |
|---|---|---|---|
| 1 | $(\mathtt{NF_a} == \mathtt{NF_b})$ | $(a_1 == b_1) \wedge (a_2 == b_2) \wedge (a_3 == b_3)$ | $(a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_3 \Leftrightarrow b_3)$ |
| 2 | $(\mathtt{NF_a} \neq \mathtt{NF_b})$ | $(a_1 \neq b_1) \vee (a_2 \neq b_2) \vee (a_3 \neq b_3)$ | $(a_1 \oplus b_1) \vee (a_2 \oplus b_2) \vee (a_3 \oplus b_3)$ |
| 3 | $(\mathtt{NF_a} > \mathtt{NF_b})$ | $(a_1 < b_1) \vee ((a_1 == b_1) \wedge (a_2 > b_2)) \vee$<br>$((a_1 == b_1) \wedge (a_2 == b_2) \wedge (a_3 > b_3))$ | $(\neg a_1 \wedge b_1) \vee ((a_1 \Leftrightarrow b_1) \wedge (a_2 \wedge \neg b_2)) \vee$<br>$((a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_3 \wedge \neg b_3))$ |
| 4 | $(\mathtt{NF_a} \geq \mathtt{NF_b})$ | $(a_1 < b_1) \vee ((a_1 == b_1) \wedge (a_2 \geq b_2)) \vee$<br>$((a_1 == b_1) \wedge (a_2 == b_2) \wedge (a_3 \geq b_3))$ | $(\neg a_1 \wedge b_1) \vee ((a_1 \Leftrightarrow b_1) \wedge (b_2 \Rightarrow a_2)) \vee$<br>$((a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2) \wedge (b_3 \Rightarrow a_3))$ |
| 5 | $(\mathtt{NF_a} \pm \mathtt{NF_b})$ | $S_1^4 \equiv [C_3, (a_1 \oplus b_1) \oplus C_2,$<br>$(a_2 \oplus b_2) \oplus C_1, (a_3 \oplus b_3) \oplus C_0]$<br>$C_1^3 \equiv (a_i \wedge b_i) \vee C_{i-1}$<br>$C_0 \equiv (`+' \Rightarrow 0) \wedge (`-' \Rightarrow 1)$ | $[(a_3 \wedge b_3) \vee ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm)),$<br>$(a_1 \oplus b_1) \oplus ((a_2 \wedge b_2) \vee ((a_1 \wedge b_1) \vee \pm)),$<br>$(a_2 \oplus b_2) \oplus ((a_1 \wedge b_1) \vee \pm),$<br>$(a_3 \oplus b_3) \oplus \pm]$ |
| 6 | $(\mathtt{NF_a} * \mathtt{NF_b})$ | $M \equiv \underbrace{\mathtt{NF_a} + \mathtt{NF_a} \ldots + \mathtt{NF_a}}_{\lvert\mathtt{NF_b}\rvert \text{times}}$<br>$m_6 \equiv a_1 \oplus b_1$ | Too large to represent<br>Apply addition ($5^{th}$ row) $\lvert\mathtt{NF_b}\rvert$ times |
| 7 | $(\mathtt{NF_a}/\mathtt{NF_b})$ | $D \equiv \underbrace{\lvert\mathtt{NF_a}\rvert - \lvert\mathtt{NF_b}\rvert - \lvert\mathtt{NF_b}\rvert \ldots - \lvert\mathtt{NF_b}\rvert}_{\#\text{times penultimate negative value}}$<br>$d_3 \equiv a_1 \oplus b_1$ | Too large to represent<br>Apply subtraction ($5^{th}$ row) D times |
| 8 | $(\mathtt{NF_a} \% \mathtt{NF_b})$ | $MOD \equiv \underbrace{\lvert\mathtt{NF_a}\rvert - \lvert\mathtt{NF_b}\rvert - \lvert\mathtt{NF_b}\rvert \ldots - \lvert\mathtt{NF_b}\rvert}_{\text{penultimate negative value}}$<br>$mod_3 \equiv 0$ | Too large to represent<br>Apply subtraction ($5^{th}$ row)<br>D ($7^{th}$ row) times |

signed bits, if we perform '3+1', the result is '4', for which 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 bit C as shown in the operation 5 of Table 1. Subtraction in a two's complement encoding is an addition with an opposite sign bit (ie., $C_0 = 1$). The multiplication pattern is described in row 6 of Table 1, which basically is a sign bit calculation plus a sequence of additions with a **double** bit-width. Division in row 7 is the times of the last but one subtraction of the second operand till the result is below zero. The modulo operation in row 8 is what is left after the division (ie., until we cannot subtract anymore keeping above zero). For multiplication and division, the sign is the XOR of the most significant bit of both operands ($a_1$ and $b_1$). The sign bit of the resulting modulo operation is always 0 (ie., modulo always returns a positive number).

The majority of SAT solvers primarily work with $\mathbb{PF}$s in CNF [9]. $\mathbb{N}$emo applies the optimal alternative – Tseitin's CNF transformation with skolemization [43]. It is the fastest known encoding to transform $\mathbb{PF}$s into a CNF formula while maintaining model equivalence and model count (ie., not altering the total number of solutions).

## 4 Nemo

Bit-blasting NFMs is a tough task to perform manually. The current prototype does it automatically including boolean and arithmetic features and constraints.

### 4.1 Prototype Overview

Fig. 1 presents an overview of Nemo, in which a *modeling expert* defines an NFM for a given SPL. Nemo provides a simple language designed to support boolean and numerical variables and mixed constraints NFMs, concretely:

- Features of domain *Boolean*, *Integer* and *Natural* (by default);
- *Constant* and *Enumerated* features, and *Ranges* of values;
- *Cardinality-based*, *Mandatory* and *Optional* (by default) features;
- *Propositional Logic*: equivalences, implications, negations, conjuctions, disjunctions, parenthetical expressions, etc.;
- *Inequalities*: equal, not equal, greater (or equal), lower (or equal); and
- *Arithmetic*: addition, subtraction, multiplication, division, and modulo.

The input to Nemo is a `.txt` file. The Nemo transformation procedure is explained in Algorithm 1.

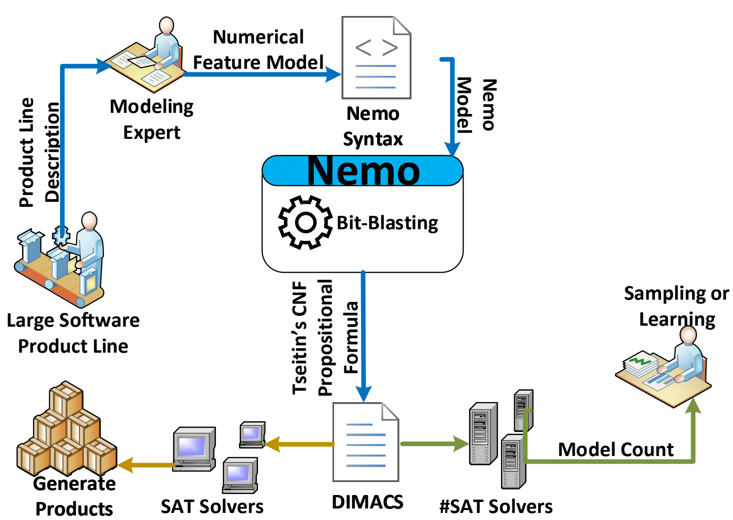


**Fig. 1.** Nemo Tool Usage Overview.

**Algorithm 1:** ℕ𝕖𝕞𝕠 Complete Procedure (blue lines of Fig. 1)

**Input:** ℕ𝔽𝕄 defined in a `.txt` file

Parse features names;
Calculate features types;
Calculate ℕ𝔽s bit-widths;
Optimize and register the declared and calculated constraints;
Bit-blast the ℕ𝔽𝕄;
Transform the bit-blasted ℕ𝔽𝕄 into a ℙ𝔽;
Transform the ℙ𝔽 into its Tseitin CNF form;
Transform the Tseitin CNF ℙ𝔽 into DIMACS;

**Result:** DIMACS file of the bit-blasted ℕ𝔽𝕄

The default output of ℕ𝕖𝕞𝕠 is an ℕ𝔽𝕄 transformed into a ℙ𝔽 in DIMACS format. DIMACS dates back to 1993 and is the de-facto input format standard for SAT solvers.[6] A DIMACS CNF file has three parts: an optional comment section with the prefix `c`, a mandatory problem line with the prefix `p`, and the clauses section following the mentioned CNF ℙ𝔽 format. `0` is a reserved keyword for a clause delimiter. DIMACS format identifies features sequentially just with a unique numerical index. Table 2 shows an example of a DIMACS file:

**Table 2.** DIMACS format example for `(A or C) and (C or not B)` formula

| Code | Description |
|---|---|
| c 1 | variable A (variables first) |
| c 2 | variable B |
| c 3 | variable C |
| p cnf 3 2 | header, CNF format, 3 variables, and 2 clauses |
| 1 3 0 | `(A or C)` (clauses last) |
| 3 -2 0 | `and (C or not B)` |

Due to our encoding, bit-vectors are identified with a name plus the sequence of bits (Big Endian); in contrast, boolean features are identified as name plus *Boolean* keyword. **Note:** A CNF Tseitin transformation of a bit-blasted ℕ𝔽𝕄 generates extra variables. Table 3 shows a bit-blasted example in DIMACS. As shown in Fig. 1, the generated DIMACS file can then be used to generate products with a SAT solver, or to count configurations with a #SAT solver. The latter is useful for fast probabilistic sampling and learning [32].

### 4.2 Numerical Feature Modeling in ℕ𝕖𝕞𝕠

Currently, most `SPL` feature modeling languages are tool-specific [33], eg., Clafer [6]. For ℕ𝕖𝕞𝕠, we abstract the notion of ℕ𝔽𝕄s defining just two entities as in [24]: generic variables and functions. Then, following the meta-model of [19], we can define a ℕ𝔽𝕄 as a formula with different domains, where a variable is a feature

[6] DIMACS: `http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability`

**Table 3.** Nemo output for expr: `(A = B) requires C; A,B` $\in$ `[-1,1], C Boolean`

| Code | Description |
|---|---|
| c 1 | `Abit1` |
| c 2 | `Abit2` |
| c 3 | `Bbit1` |
| c 4 | `Bbit2` |
| c 5 | `Tseitin1` |
| c 6 | `Tseitin2` |
| c 7 | `Tseitin3` |
| c 8 | C Boolean |
| p cnf 8 14 | header, cnf format, 8 variables, and 14 clauses |
| -2 0 | `not Abit2` |
| -4 0 | `not Bbit2` |
| -2 -4 -5 0 | `and (not Abit2 or not Bbit2 or not Tseitin1` |
| 2 4 -5 0 | `and (Abit2 or Bbit2 or not Tseitin1)` |
| 2 -4 5 0 | `and (Abit2 or not Bbit2 or Tseitin1)` |
| -2 4 5 0 | `and (not Abit2 or Bbit2 or Tseitin1)` |
| -1 -3 -6 0 | `and (not Abit1 or not Bbit1 or not Tseitin2)` |
| 1 3 -6 0 | `and (Abit1 or Bbit1 or not Tseitin2)` |
| 1 -3 6 0 | `and (Abit1 or not Bbit1 or Tseitin2)` |
| 3 -1 6 0 | `and (Bbit1 or not Abit1 or Tseitin2)` |
| -6 7 0 | `and (not Tseitin2 or Tseitin3)` |
| -5 7 0 | `and (not Tseitin1 or Tseitin3)` |
| 5 6 -7 0 | `and (Tseitin1 or Tseitin2 or not Tseitin3)` |
| 8 7 0 | `and (C or Tseitin3)` |

and a function is a hierarchical relationship or constraint. For that, we decided to define a keywords-based syntax for our first prototype. Our motivation was to reduce Nemo's learning curve. Consequently, we used a cheat sheet:

- `def Var_Name D` defines a named feature with `D` as its domain or range;
- `bool`, `integer` and `natural` (ie., natural numbers or positive integers) are the supported domains;
- `[X]` indicates a constant feature with a value `X`;
- `[X:Y]` indicates a range between `X` and `Y` inclusive;
- `[X,Y,Z]` indicates an enumerated type with restricted values `X`, `Y` or `Z`;
- `and`/`or` are the conjunctions and disjunctions;
- `<->`/`->`/`neg` are equivalences, implications and negations;
- `=`/`>`/`<`/`>=`/`<=`/`!=` are the equalities/inequalities; and
- `+`/`-`/`*`/`/`/`%` are the numerical operators.

Listing 1.1 illustrates most of the types of supported clauses:

```
def A_constant [3]
def B_natural [0:3]
def C_natural_2 [:3]
def D_integer [-2:1]
def E_enumerated_integer [-1, 2, 4, 8]
def F_new_Boolean bool 0
def G_predefined_Boolean bool 23
ct  G_predefined_Boolean or F_new_Boolean
ct  ((A_constant * B_natural) > C_natural) ->
    (F_new_Boolean or (E_enumerated_integer = D_integer))
```

**Listing 1.1.** A ℕemo ℕ𝔽𝕄: `(G or F) and ((A*B)>C) requires (F or (E = D))`

Sequentially, the above means:

1. `A_constant`: a constant `natural` ℕ𝔽 with a value of `3`;
2. `B_natural`: a `natural` ℕ𝔽 between `0` and `3`;
3. `C_natural_2`: a `natural` ℕ𝔽 between `0` and `3`,
4. `D_integer`: an integer ℕ𝔽 between −`2` and `1` in two's complement encoding;
5. `E_enumerated_integer`: an enumerated integer ℕ𝔽 with exactly 4 values;
6. `F_new_Boolean`: a boolean feature. Zero (`0`) means that it is a new feature;
7. `G_predefined_Boolean`: a boolean feature defined in a previous DIMACS ℕ𝔽𝕄 where `23` is that feature identifier in the original ℕ𝔽𝕄;
8. A boolean parenthetical propositional expression: `(G or F)`; and
9. An arithmetic constraint: `((A*B)>C) requires (F or (E = D))`.

We have two tags for the objects: `def` are feature declarations and `ct` are their constraints. The format is flexible, allowing any tag at any line. Range definitions can have one of the limits omitted (eg.,[: 3] is considered as [`0` : `3`]).

### 4.3 Implementation of Smart Transformations

ℕemo is a cross-platform tool developed in Python 3.10.8 x86_64. We tackled several engineering challenges in its implementation.

First, ℕemo dynamically sets a feature as a `natural` or an `integer`, as the bit-blasted encoding of some operations are different (ie., inequalities, division, and modulo).[7] If any value of a ℕ𝔽 is negative, it is considered an `integer`.

Second, ℕemo dynamically calculates the minimum bit-width of each ℕ𝔽 to generate the shortest ℙ𝔽. The process is based on the possible values of each ℕ𝔽 (eg., range, enumeration) and the domain; `natural` ℕ𝔽s and constraints produce smaller ℙ𝔽s. For instance, the most optimal encoding for an enumerated feature with just two values (eg., -1 and 9), and that is not involved in arithmetic expressions, is a single bit `natural` ℕ𝔽.

Third, ℕemo readjusts the previous computed widths based on ℕ𝔽𝕄 constraints. Leaving aside boolean features, every ℕ𝔽 involved in operations with other ℕ𝔽s must have the same type and bit-width in order to apply bit-blasting.

[7] Besides inequalities, division, and modulo, arithmetic operations do not make unsigned/signed distinction due to the Two's complement encoding.

Our solution was to recursively search for the NF with the highest bit-width of each set of NFs involved in a constraint, and set that bit-width to the rest of the features sharing a constraint. For instance, transforming a `natural` into an `integer` NF, is to add one bit for the sign.

Fourth, Nemo readjusts bit-widths in case of mathematical operations that can produce extra carry-bits. The most efficient is to define the highest from:

- **Addition:** Extending one bit for the first addition, followed by extra bits per sets of two extra additions. For example, `A + B + C + D = E` needs two extra carry bits. Note that `natural` numerical ranges are up to $2^{\texttt{bit-width}}-1$.
- **Multiplication:** The extended bit-width is the original multiplied by the number of multiplication operands plus `1`. For instance, `A * B * C = D` implies that $\texttt{bit-width}_{\texttt{updated}} = (\texttt{bit-width}_{\texttt{current}} \times 3){+}1$.

### 4.4 Nemo Optimizations by Pre-Processing the NFM

Bit-blasting and Tseitin transformations create different size CNF PFs depending on the equation. Nemo takes advantage of that by replacing and adjusting constraints to produce shorter PFs. Concretely:

1. `>`/`<`/`+`/`-` do not create extra variables;
2. $\geq$/$\leq$ create (*bit-width*−`1`) Tseitin variables in the NFs involved;
3. `=` creates (*bit-width*) Tseitin variables in the NFs involved;
4. $\neq$ creates (*bit-width*+`1`) Tseitin variables in the NFs involved;
5. `/` creates ($3{\times}2^{\texttt{bit-width}-1}$) Tseitin variables in the NFs involved;
6. `%` creates ($14{\times}2^{\texttt{bit-width}-1}$) Tseitin variables in the NFs involved; and
7. `*` creates ($6^{\texttt{bit-width}-1}$) Tseitin variables in the NFs involved.

The only two operations naturally replaceable by an alternative with a shorter PF encoding are {$\geq$,$\leq$} by {`>`,`<`} respectively. (eg., `A`$\geq$`1` and `A`$\leq$`2` are equivalent to `A>0` and `A<3`). Additionally, Nemo removes duplicated constraints. For example, in case of the constraints `A<2` and `A<1` the first one is redundant. Finally, Nemo dynamically prioritizes `natural` NFs, as unsigned operations are more scalable – need smaller bit-widths and produce smaller PFs.

## 5 Evaluation

We answer the following research questions to evaluate Nemo:

**RQ1**: Are Nemo bit-blasted NFMs viable for any bit-width?
**RQ2**: Do Nemo bit-blasted NFMs allow faster counting?

**RQ1** evaluates the viability of Nemo for different NFM constraints with increasing bit-widths. **RQ2** evaluates how Nemo performs compared to state-of-art SMT and CP solvers for large real-world `SPL`s. To count the number of configurations of Nemo bit-blasted NFMs, we used `sharpSAT` [42], the state-of-the-art model counter for ℙ𝔽s. Every test has been carried out on an Intel(R) Core i7-4790 CPU@3.60 GHz processor with 16 GB of memory RAM and an SSD running an up-to-date Lubuntu 20.04 LTS X86_64.

### `RQ1`: Are ℕemo bit-blasted ℕ𝔽𝕄s viable for any bit-width?

We start analyzing the most complex types of ℕ𝔽𝕄 operations – arithmetic. Additionally, we add the least complex inequality (i.e., `=`), which allows us to focus on arithmetic equalities. For similar reasons, we opted for `natural` instead of `integer` ℕ𝔽s. The first set of 5 ℕ𝔽𝕄 constraints that were analyzed are defined by (`(A op B) = C`) where `op` ∈ `{+,-,*,/,%}` from now on.

Formulas with different bit-widths (#`b`) from 2 up to 16 step 2 were generated. Remember that the maximum bit-width, as said earlier, is limited to the most demanding operation. Finally, if counting surpasses `15` minutes we considered it a *time-out* due to a high probability of never finishing. For each expression, we measured: a) the number of CNF clauses generated in each ℙ𝔽, and b) the time in seconds to count the configurations of those ℙ𝔽s with `sharpSAT`.

Fig. 2 shows in two graphs the first results. The X-axis are bit-vector widths, and the Y-axis is the number of generated clauses or counting time in seconds respectively. As operation `*` counting timed-out, we scale the graphs up to bit-width 16. It is worth noting that Fig. 2 was truncated at 16-bits, even though addition (`+`) did not time-out even when the bit-width was 40.

- Number of clauses: `+` and `-` linearly grow in direct proportion with the bit-width. `/` and `%` almost linearly grow at 2× rate. `*` grows exponentially.
- Time to count: `+` and `-` linearly grow in proportion with the bit-width, keeping below a second for 16 bit-width. `*`, `/` and `%` grow exponentially, keeping below 50 seconds for 12 bit-width.

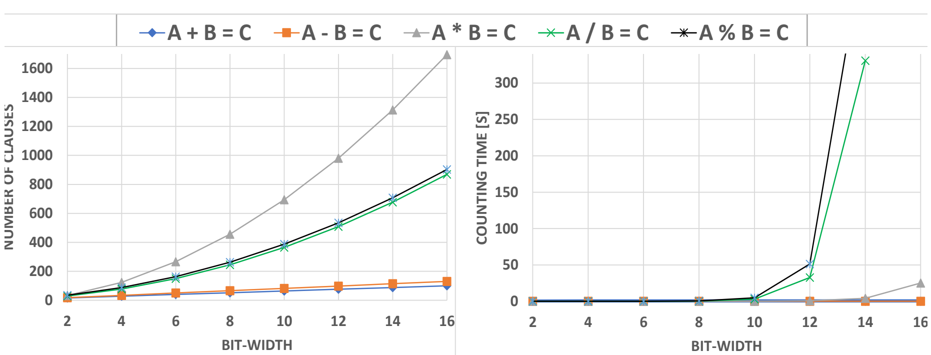


**Fig. 2.** ℕemo generated clauses and counting time in seconds of arithmetic operations.

As the number of ℕ𝔽 variables is proportional to the bit-width, Tseitin's transformation guarantees a linear increase `O(3n+1)` [43]. As operation `*` creates

($2{*}bit\text{-}width$) carry-bits, Tseitin's transformation increase is negligible. Hence, carry-bits are the ones causing the exponential growth.[8]

Further, we analyze logic and arithmetic mixed nested constraints, and up to four conjuncted numerical constraints. Following the previous procedure, we prioritize the fewer demanding operations (ie., `=`, `+`, $\Rightarrow$) to reduce interactions for cleaner insights. The second set of 4 constraints analyzed are:

1. ((A + B) = C) $\Rightarrow$ D
2. (A + B) = C
3. (A + B) = C $\wedge$ (D + E) = F
4. ((A + B) = C)$\wedge$((D + E) = F)$\wedge$((G + H) = I)$\wedge$((J + K) = L)

Fig. 3 shows the second set of results. Again, the number of clauses is linearly proportional to the bit-width and the number of operations and constraints. Boolean operations needed fewer clauses than arithmetic operations. While nesting did not especially affect the number of clauses, it caused an exponential increase in counting time including a 12 bit-width time-out.

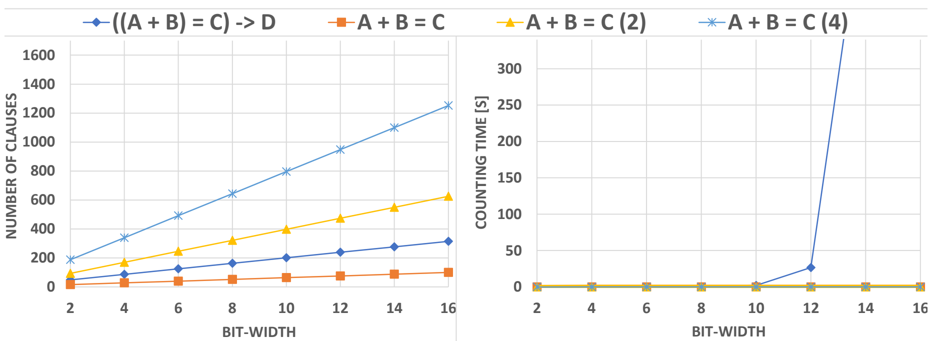


**Fig. 3.** Nemo generated clauses and counting time in seconds of sets of equations.

Nemo **natural** encoding was feasible up to a 12 bit-width. While multiplication is the only exponential transformation, division, modulo, and nesting are the slowest to count; we suggest discretizing their numerical ranges, keeping them within 12 bit-width for the current prototypes of Nemo and `sharpSAT`.

***Conclusion:* *Nemo NFs are unbounded by default, but its encoding scales up to 12 bit-width per number with a counting time under 10 seconds. The number of clauses is indirectly related to that time. Division, modulo, and nesting are the slowest constraints to count.***

[8] Multiplying two bit-vectors could generate a double width one (eg., $2^3 * 2^3 = 2^6$). Those are many carry-bits compared to additions which create a maximum of one.

### RQ2: Do $\mathbb{N}$emo bit-blasted $\mathbb{NFM}$s allow faster counting?

We expect **RQ1** counting conclusions to be an upper-bound for other solvers as larger bit-widths imply more configurations to analyze, which sometimes becomes exponential. We used 7 real-world $\mathbb{NFM}$s obtained from [29] and [37]. Table 4 lists them, where each system has a different number of $\mathbb{NF}$s and/or different configuration space size. Henceforth, we use FSE2015 to denote the $\mathbb{FM}$s from [37]. We modeled their $\mathbb{NFM}$s independently in Clafer, Z3, and $\mathbb{N}$emo syntax, and additionally ran $\mathbb{N}$emo to generate the $\mathbb{PF}$s just for `sharpSAT`. Considering the bottleneck found in **RQ1**, if a $\mathbb{NF}$ is unbounded or surpasses $[0, 2^{12}-1]$, it is limited to $2^{12}$ options for the 3 solvers, which implies values discretization within some of the $\mathbb{NFM}$s (eg., we can represent the even values of $[0, 2^{13}]$ in 12 bits).

**Table 4.** List of Models with Numerical Features and Constraints

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

**Table 5.** Clafer, Z3 and $\mathbb{N}$emo encoding `sharpSAT` counting time for 7 real-world SPLs

| Type | **Model** | **Z3** | **Clafer** | `sharpSAT` |
|---|---|---|---|---|
| FSE2015 | Dune | 26.18 s | 10.49 s | 0.01 s |
| | HSMGP | 40.70 s | 13.91 s | 0.01 s |
| | HiPAcc | 457 s | 32.52 s | 0.01 s |
| | Trimesh | Time-out | 217.01 s | 0.01 s |
| KConfig | axTLS | Time-out | Time-out | 0.01 s |
| | Fiasco | Time-out | Time-out | 0.01 s |
| | uClibc-ng | Time-out | Time-out | 0.01 s |

We compared for the same number of solutions the time to count them in seconds with `sharpSAT`, and one CP and SMT solvers: Clafer[9] and Z3[10] respectively. Z3 and Clafer do not strictly perform model counting as sharpSAT does, instead they enumerate by: 1) deriving a configuration, 2) making the negation of that solution as a constraint, and 3) repeating steps 1 and 2 until the con-

[9] Clafer: `https://www.clafer.org/`
[10] Z3py: `https://github.com/Z3Prover/z3`

strained model is unsatisfiable.[11,12] If counting took more than 15 minutes, we considered it a *time-out*. Table 5 lists the results. In summary, `sharpSAT` counted NFMs configurations in under 0.01 seconds where Z3 and Clafer timed-out for `KConfig` models. This empirically demonstrates the superior speed of algorithms for model counting versus enumerating.

***Conclusion: `sharpSAT` counts the configurations of* Nemo PFs *considerably faster than Z3 and Clafer do with native* NFMs*.***

## 6 Nemo Tool Scalability and Threats to Validity

In **RQ1** we tested the scalability of counting with `sharpSAT`the bit-blasted models generated by Nemo. In this section, we use the same NFMs to test the scalability of the transformation process itself. In other words, we now evaluate Nemo's runtime. We present the results in Fig. 4, and although there are two tools performing different tasks, we infer similar conclusions. Nemo finishes *instantly* for addition and subtraction operations. However, the runtime time is slightly exponential for division and modulo, and truly exponential for multiplication, due to the carry bits of the operations. Nevertheless, all transformations finished below 40 minutes for a 16 bit-width. Regarding nested and stacked constraints, it takes a maximum of 85 seconds to process all equalities. Comparing Fig. 2 and Fig. 4, there is clear relationship between Nemo transformation time and the number of clauses of that transformation. This scalability issue was theoretically predicted [11].

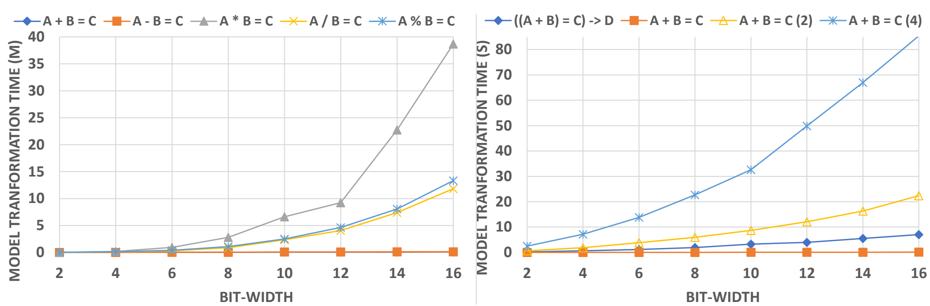


**Fig. 4.** Nemo runtime in seconds of arithmetic operations and equations sets.

**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 [40]. For **RQ2**, we used the counting methods that are proposed by the developers of each solver: `sharpSAT` is the default execution, Clafer requires the `noprint` option, and Z3 requires a counting loop.

[11] Z3 developer on model counting: `https://github.com/Z3Prover/z3/issues/934`

[12] Clafer Choco solver: `https://github.com/chocoteam/choco-solver/blob/master/src/main/java/org/chocosolver/solver/search/strategy/Search.java`

**External validity.** We used the 7 real-world SPLs of Table 4 which have different numbers of features, domains, and constraints. For complex constraints, synthetic models were evaluated. While we are aware that our results may not generalize to all SPLs, their trends are identical in different cases. Similarly, although being state-of-the-art, Z3 and Clafer may not be representative of all SMT and CP solvers. Additionally, a manual bit-blasting approach for $\mathbb{NF}$s and basic operations was successfully applied for counting-based optimizations of SPLs [26]. Our work extends the encoding to complete arithmetic, and creates a tool that allows $\mathbb{NFM}$s modeling while automatizing and optimizing all the process.

## 7 Related Work

Work tackling $\mathbb{NFM}$s is rare [22]. Some considered $\mathbb{NF}$s as classical features with just present/absent states [8,29,15]. Some encoded $\mathbb{NF}$s as alternative features, where each value of a $\mathbb{NF}$ was considered a distinct feature [20]. Shi [36] used a single type of feature called 'pseudo-boolean' with only `Successor` (`+1`) and `Predecessor` (`-1`) operations. In [7], each boolean feature had related attributes – a set of variables in the form (name, value, domain). However, attributes and $\mathbb{NF}$s are essentially different: attributes are not nodes of the variability tree, and as opposed to a $\mathbb{NF}$, a change in the value of an attribute does not result in a different configuration [25]. Hence, counting the size of a product space will return a lower-than-expected value.

SMT and CP solvers natively support representation and reasoning of $\mathbb{NFM}$s. However, #CP or #SMT solvers, counting generalizations of CP and SMT, are nonexistent. This is to be expected, as CP and SMT theories are unbounded by default [31], being unaware of allocated memory or domain definitions (eg., undefined maximum of `x` in `x≥1`). In SAT theory, all variables are bounded (ie., boolean). Consequently, SMT approximation counting has been proposed [14]. STP solver [16] implements a bit-vector approach for counting. It performs array optimizations and arithmetic and Boolean simplifications before bit-blasting to MiniSat [38]. While it works to test satisfiability by counting at least one, it does not preserve counting or model equivalence. This is in line with the most recent model counting competition (2020), where 34 versions of 8 fastest counting solvers were tested. Model counting is more commonly found in *Binary Decision Diagrams* [12] and SAT-based [42] solvers. The results indicate that while fast, even so-called 'exact solvers' count a close but inexact number of configurations.

Simplification of $\mathbb{NFM}$s usually reduces reasoning time. However, those beyond the ones implemented in $\mathbb{N}$emo do not preserve counting or model equivalence [13]. Nevertheless, the bit-width bottleneck is shared even in solutions that perform approximate counting. An example is Boolector reasoner [10], which lazily instantiates array axioms and macros. Even Z3 [23] applies bit-blasting to every operation besides equality, which are, then, handled by a specific algorithms.

## 8 Conclusions and Future Work

The size of an SPL configuration space grows exponentially with an increasing number of features. Compared to classical FMs, NFMs have more complex relationships due to larger domains (`natural` and `integer`) and more complex types of constraints (ie., arithmetic). That makes techniques of statistical reasoning and learning that much more important to understand and support, where a key reasoning operation is model counting. Unfortunately, while automated solvers can analyze NFMs, they were not developed with the objective of counting configurations. Again, counting configurations is key to finding near-optimal SPL configurations (eg., find one of the top configurations minimizing the runtime of a given benchmark [26,28,32]).

We developed Nemo, a prototype that automatically optimizes and transforms NFMs to a Tseitin PF in DIMACS format. Nemo represents NFs as bit-vectors by means of bit-blasting, while arithmetic constraints are encoded as propositional clauses. We evaluated Nemo by transforming different synthetic and real-world NFMs to PFs and used existing SAT-based approaches to count configurations. We have shown that Nemo can:

- model, automatically optimize, and transform NFMs by using the Nemo language;
- use bit-blasting to encode common types of numerical features and arithmetic constraints;
- represent NFMs up to 12 bit-width of accuracy without overhead for almost every combination of boolean and arithmetical operations;
- use `sharpSAT` to count the number of configurations up to $10^{45}$ products in under 0.01 seconds. Analyzing a $10^{45}$ product space is infeasible with current state-of-the-art SMT and CP solvers as they count by enumeration.

We are confident our work can support statistical and learning techniques that analyze NFMs of real-world SPLs. Our research also suggests future explorations:

- bit-blast more features of other domains and with new types of relationships (eg., strings with concatenation and sub-string operations);
- optimize the transformation to generate models that are faster to count;
- run Nemo in an ecosystem with different solvers with extended support (eg., attributes, graphical interface); and
- beautify Nemo's language to be a more human-friendly modeling language.

**Acknowledgements.** Munoz, Pinto and Fuentes work is supported by the European Union's H2020 research and innovation programme under grant agreement DAEMON 101017109, by the projects co-financed by FEDER funds LEIA UMA18-FEDERJA-15, MEDEA RTI2018-099213-B-I00 and Rhea P18-FR-1081 and the PRE2019-087496 grant from the Ministerio de Ciencia e Innovación. Batory is retired, writing free textbooks [5], and is walking dogs for wages.

## References


1. Agh, H., García, F., Piattini, M.: A checklist for the evaluation of software process line approaches. Information and Software Technology **146**(1) (sep 2022)
2. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-oriented Software Product Lines. Springer, NY, USA (2016)
3. Barrett, C.: Decision Procedures: An Algorithmic Point of View, Springer-Verlag, 2008. Journal of Automated Reasoning **51**(4) (2013)
4. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Model Checking. Springer, NY, USA (2018)
5. Batory, D.: Automated Software Design, Vol. 1. Lulu.com (2021)
6. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In: SLE (2010)
7. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems **35**(6) (2010)
8. Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K.: A Study of Variability Models and Languages in the Systems Software Domain. IEEE TSE **39**(12) (2013)
9. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. IOS press, IEEE (2009)
10. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-vectors and Arrays. In: TACAS. Springer, NY, USA (2009)
11. Bryant, R., et al.: Deciding Bit-vector Arithmetic with Abstraction. In: TACAS. Springer, NY, USA (2007)
12. Bryant, R.: Binary Decision Diagrams. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer (2018)
13. Chakraborty, S., et al.: Approximate model counting. FRONTIERS (2021)
14. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate Counting in SMT and Value Estimation for Probabilistic Programs. Acta Informatica **54**(8) (2017)
15. Döller, V., Karagiannis, D.: Formalizing Conceptual Modeling Methods with MetaMorph. In: Enterprise, Business-Process and Information Systems Modeling. Springer, Springer (2021)
16. Ganesh, V., Dill, D.: The Simple Theorem Prover (STP) solver. https://stp.github.io/ (2006)
17. Henard, C., Papadakis, M., Harman, M., Traon, Y.L.: Combining Multi-objective Search and Constraint Solving for Configuring Large Software Product Lines. In: SPLC. IEEE Press, NJ, USA (2015)
18. Horcas, J., Pinto, M., Fuentes, L.: Variability Models for Generating Efficient Configurations of Functional Quality Attributes. IST Journal **95** (2018)
19. Horcas, J., Pinto, M., Fuentes, L.: Extensible and modular abstract syntax for feature modeling based on language constructs. In: SPLC. ACM (2020)
20. Kästner, C., et al.: Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: OOPSLA. vol. 46. IEEE/ACM, NJ, USA (2011)
21. Liang, J., Ganesh, V., Czarnecki, K., Raman, V.: SAT-based Analysis of Large Real-world Feature Models Is Easy. In: SPLC. IEEE/ACM, NJ, USA (2015)
22. Marchezan, L., Rodrigues, E., Assunção, W.K.G., Bernardino, M., Basso, F.P., Carbonell, J.: Software product line scoping: A systematic literature review. Journal of Systems and Software **186** (2022)
23. Moura, L.D., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. Springer, NY, USA (2008)

24. Munoz, D., et al.: Category Theory Framework for Variability Models with Non-functional Requirements. In: CAiSE. Springer International Publishing (2021)
25. Munoz, D.J., Pinto, M., Fuentes, L.: Finding correlations of features affecting energy consumption and performance of web servers using the hadas eco-assistant. Computing **100**(11), 1155–1173 (2018)
26. Munoz, D., Oh, J., Pinto, M., Fuentes, L., Batory, D.: Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In: SPLC (2019)
27. Munoz, D., Pinto, M., Fuentes, L.: HADAS: Analysing Quality Attributes of Software Configurations. In: SPLC. SPLC '19, ACM (2019)
28. Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding Near-optimal Configurations in Product Lines by Random Sampling. In: ESEC/FSE (2017)
29. Oh, J., Gazillo, P., Batory, D., Heule, M., Myers, M.: Uniform Sampling from Kconfig Feature Models. Tech. Rep. TR-19-02, University of Texas at Austin, Department of Computer Science (2019)
30. Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product Sampling for Product Lines: The Scalability Challenge. In: SPLC. SPLC '19, Association for Computing Machinery (2019)
31. Phan, Q.: Model Counting Modulo Theories. Ph.D. thesis, Queen Mary University of London (apr 2015)
32. R. Heradio, D. Fernandez-Amoros, J. Galindo, D. Benavides, D. Batory: Uniform and Scalable Sampling of Highly Configurable Systems. Empirical Software Engineering (Jan 2022)
33. Raatikainen, M., Tiihonen, J., Mannisto, T.: Software Product Lines and Variability Modeling: A Tertiary Study. Journal of Systems and Software **149** (2019)
34. Rossi, F., Beek, P.V., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
35. Schmitt, A., Wiersch, S., Weis, S.: Glencoe-a Visualization Prototyping Framework. In: ICCE (2015)
36. Shi, K.: Combining Evolutionary Algorithms with Constraint Solving for Configuration Optimization. In: ICSME. IEEE/ACM (sep 2017)
37. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence Models for Highly Configurable Systems. In: FSE. ACM, New York, NY, USA (2015)
38. Sorensson, N., Een, N.: Minisat V1. 13-a Sat Solver with Conflict-clause Minimization. SAT **2005**(53) (2005)
39. Sundermann, C., Nieke, M., Bittner, P.M., Heß, T., Thüm, T., Schaefer, I.: Applications of #SAT Solvers on Feature Models. In: VaMoS. ACM, NY, USA (2021)
40. Systems, C.: Sample Size Calc. `https://www.surveysystem.com/sscalc.htm`
41. Thüm, T., et al.: FeatureIDE: An Extensible Framework for Feature-oriented Software Development. Science of Computer Programming **79** (2014)
42. Thurley, M.: SharpSAT–counting Models with Advanced Component Caching and Implicit BCP. In: SAT. Springer Berlin Heidelberg (2006)
43. Tseitin, G.: On the Complexity of Derivation in Propositional Calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970. Springer Berlin Heidelberg (1983)