Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language[☆]

Daniel-Jesus Munoz[a,*], Mónica Pinto[a], Lidia Fuentes[a], Don Batory[b]

[a] ITIS Software, Universidad de Malaga, Andalucia Tech, Bulevar Louis Pasteur 35, 29010, Malaga, Spain
[b] Department of Computer Science, University of Texas at Austin, TX 78712, USA

ARTICLE INFO

Article history:
Received 3 December 2022
Received in revised form 8 April 2023
Accepted 29 May 2023
Available online 3 June 2023

Keywords:
Feature model
Bit-blasting
Propositional formula
Numerical features
Model counting
Universal variability language

ABSTRACT

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

ℕ𝕖𝕞𝕠 (Numbers, features, models) is a tool that supports ℕ𝔽𝕄s by *bit-blasting*, the technique that encodes arithmetic expressions as boolean clauses. The newest version, ℕ𝕖𝕞𝕠2, translates ℕ𝔽𝕄s to propositional formulas and the *Universal Variability Language* (UVL). By doing so, products can be counted efficiently by #SAT and Binary Decision Tree solvers, enabling finding near-optimal products. This article evaluates ℕ𝕖𝕞𝕠2 with a large set of synthetic and colossal real-world ℕ𝔽𝕄s, including complex arithmetic constraints and counting and sampling experiments. We empirically demonstrate the viability of ℕ𝕖𝕞𝕠2 when counting and sampling large and complex SPLs.

1. Introduction

Software Product Line(SPL) engineering is a key reuse approach to build highly-configurable systems (Agh et al., 2022). 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* (Apel et al., 2016; Batory, 2021). The set of all legal configurations is the SPL's *product space*.

Real-world SPLs need *Numerical Feature Models(ℕ𝔽𝕄s)*. One of many examples is the SPL of Linux repositories where packages have different versions and other numerical attributes, called *Numerical Features(ℕ𝔽s)* (Oh et al., 2019). Relationships among ℕ𝔽s are arithmetic constraints. In effect, ℕ𝔽𝕄s are 𝔽𝕄s with ℕ𝔽s.

SAT solvers efficiently find configurations of classical 𝔽𝕄s, because 𝔽𝕄s can be translated to ℙ𝔽s, and SAT efficiently finds ℙ𝔽 solutions (ie., configurations). Unfortunately, SAT performs poorly on counting as it enumerates products, which is infeasible for large SPL product spaces, $\geq 10^6$ products (Pett et al., 2019).

Why is counting important? Because counting products enables unbiased random samples on large product spaces (Liang et al., 2015; Oh et al., 2017). This enables near-optimal configurations to be located in an SPL product space with statistical guarantees (eg., x% from optimal with y% confidence), given a defined workload (Oh et al., 2017; Sundermann et al., 2021c; Oh et al., 2024).

Only a handful of automated solvers support ℕ𝔽𝕄s, namely *Satisfiability Modulo Theories(SMT)* (Barrett and Tinelli, 2018) and *Constraint Programming(CP)* (Rossi et al., 2006) solvers. Unfortunately, SMT and CP solvers perform brute-force enumeration to count (Munoz et al., 2022). In contrast, #SAT solvers extend SAT solvers to count the number of solutions of a ℙ𝔽 efficiently without enumeration (Biere et al., 2009). #SAT solvers out-perform SMT and CP solvers on counting. Likewise, *Binary Decision Tree(BDD)* solvers outperformed other solvers when uniformly random sampling product spaces of any size (Heradio et al., 2022).

We use techniques to translate ℕ𝔽𝕄s into ℙ𝔽s (Munoz et al., 2019a). Concretely, *bit-blasting* (Bryant et al., 2007) encodes numerical values into bits and arithmetic constraints into ℙ𝔽s.

This article is an invited extension of Munoz et al. (2022), where we presented ℕ𝕖𝕞𝕠*(Numbers, features, models)*, a tool that natively supports ℕ𝔽𝕄s and efficient SAT operations to find ℕ𝔽𝕄 products (satisfying boolean and arithmetic constraints) as well

[☆] Editor: Laurence Duchien.
[*] Corresponding author.
E-mail addresses: dm@uma.es (D.-J. Munoz), mpinto@uma.es (M. Pinto), lfuentes@uma.es (L. Fuentes), batory@cs.utexas.edu (D. Batory).

https://doi.org/10.1016/j.jss.2023.111770

as #SAT counting NFM products. In this work, we present Nemo2, an extension with more functionality, such as new input and output formats like *Universal Variability Language(UVL)* (Sundermann et al., 2021s) models that are compatible with different state-of-the-art solvers like BDDs. Additionally, Nemo2 now extends/composes already modeled FMs with new NFs. Nemo2's NFM grammar is simple; it supports constant, enumerated, and range variables, along with boolean and arithmetic constraints. Given an NFM, Nemo2 generates two types of PFs or an UVL model, as they are standard formats for many tools like SAT or BDD-based ones. At this point, we can invoke SAT, #SAT or BDD automated reasoners.

The novel contributions of our paper are:

- Explaining how Nemo2 automatically translates and optimizes the encoding of arithmetic operations (as complex as multiplication, division, and modulo) and arithmetic constraints on NFs into classical PFs, Tseitin CNF PFs and UVL models;
- Experimentally testing the viability of Nemo2 with a large set of synthetic and 12 colossal real-world NFMs up to a configuration space size of $\sim 5.66 \times 10^{1953}$.
- Experimentally testing the viability of Nemo2 with complex arithmetic constraints and space sizes when counting and sampling with the current state-of-the-art solvers Glucose3, sharpSAT, FlamaPY BDD and BDDSampler.

Nemo2 is open-source and available in GitHub and Zenodo. [1]

2. Bit-blasting background and overview

2.1. Propositional formulas of feature models

A classical feature model defines every feature of an SPL, along with constraints. Features are notoriously dependent. That is, selecting one feature may preclude or require many other features. It is well-known we can translate classical feature models into a PF ϕ, where features are boolean variables, and constraints are clauses. This enables off-the-shelf SAT and #SAT technologies to analyze feature models, such as finding dead code and performing safe composition (Apel et al., 2016; Batory, 2021; Benavides et al., 2007; Czarnecki and Pietroszek, 2006). State-of-the-art tools that convert feature models into PFs are

FeatureIDE (Thüm et al., 2014) and Glencoe (Schmitt et al., 2015); both translate a graphically drawn feature model into a PF in *Conjunctive Normal Form(CNF)*.

2.2. Finding near-optimal configurations

If an SPL has $\boldsymbol{f}$ binary features without constraints, its product space $\mathbb{C}$ is of size $|\mathbb{C}|=2^{\boldsymbol{f}}$. When $\boldsymbol{f}=80$, which is small for many SPLs, 2^{80} equals 10^{24}, the estimated number of stars in the universe.

A core problem in SPL usage is to find a near-optimal configuration in an SPL product space. Searching configuration spaces by enumeration is possible only for minuscule $\mathbb{C}$. Using uniform random sampling (where each configuration has the same probability of being selected), we can quickly search in colossal configuration spaces (exceeding 10^{1440} in size) for near-optimal configurations. The entire approach to uniformly random sample configurations is based on counting the number of configurations in a product space.

Given a random integer j in $[1..|\mathbb{C}|]$, the trick is to convert j into the jth configuration in $\mathbb{C}$. This is done by a binary search by choosing a feature f and counting the size of the space of configurations with f. If $j \leq |\phi \wedge f_i|$, the jth configuration has feature f, recurse on the space $(\phi \wedge f)$, otherwise the jth configuration has feature $\neg f$ and recurse on $(\phi \wedge \neg f)$. At each iteration, a new feature is chosen, counting is performed, and the features belonging to the jth configuration is eventually found.

The algorithm returns the best-performing configuration ¢ in a sample of size n, requiring in $n \cdot f$ calls to a counting (#SAT) tool. Configuration ¢ is *on average* $\frac{100}{n+1}$ percentiles away from *the* best-performing configuration in $\mathbb{C}$ with $\frac{100}{n+1}$ percentiles standard deviation. So, if 99 uniformly random samples are taken, the best-performing configuration out of the 99 is an average 1%±1% away from the best configuration in $\mathbb{C}$. This holds for arbitrary-large configuration space. Details are in Oh et al. (2017, 2024).

2.3. Numerical features

However, real-world SPLs use NFMs that contain both binary features and NFs (Henard et al., 2015). An NF has a name N, a type (ie., domain), and range (eg., $N \in [1, 2, \ldots, 128]$). NFMs add arithmetic constraints to the set of propositional connectives. Further, arithmetic constraints can negate or assert boolean feature values and vice-versa.

Two examples of NFMs are: (1) the HADAS eco-assistant (Munoz et al., 2019) where energy parameters are coded by NFs in an integer domain, and propositional connectives and inequalities arise in cross-tree constraints (eg., $\mathit{AEScrypto} \Rightarrow \mathit{keySize} > 128$) and (2) WeaFQAs (Horcas et al., 2018) has integer and float attributes with propositional connectives and interval constraints (ie., numerical value ranges).

2.4. Bit-blasting

Bit-blasting, also called *flattening*, is the transformation of a bit-vector arithmetic formula into a PF (Barrett, 2013). Variables are bit-vectors, and arithmetic operations are propositional clauses that reference bits. The resulting PF is satisfiable whenever the original arithmetic formula is. Our work focuses on basic arithmetic relations and operations and counting. We present operations in order of their usage frequency in real-world NFMs (Munoz et al., 2019a): equality (=), inequalities (=/, >, $\underline{>}$), addition (+), subtraction (−), multiplication (∗), division (/), and modulo (%).

2.5. Bit-blasting basic arithmetic operations

The main property of bit-vectors is their width which defines: (a) the minimum and maximum value limits of numerical variables, and (b) whether the vector is unsigned (ie., binary *sign-magnitude* encoding) or signed (ie., binary *two's complement* encoding).[2] We use the Big-Endian representation[3] where the first bit of the bit-vector encodes the sign as positive (0) or negative (1).

Table 1 has examples of two's complement bit-blasting PFs for arithmetic relations on Big-Endian signed integers with a value

[1] Nemo2 can be downloaded from:
- https://github.com/danieljmg/Nemo2_tool
- https://doi.org/10.5281/zenodo.7780854

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

[3] 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 The sign bits are a_1 and b_1 meaning that a value of 1 represents a negative number.

Row	Operation	Bit-Blasted model	Propositional formula
1	$(\mathtt{NF_a} == \mathtt{NF_b})$	$(\mathtt{a_1} == \mathtt{b_1}) \wedge (\mathtt{a_2} == \mathtt{b_2}) \wedge (\mathtt{a_3} == \mathtt{b_3})$	$(\mathtt{a_1} \Leftrightarrow \mathtt{b_1}) \wedge (\mathtt{a_2} \Leftrightarrow \mathtt{b_2}) \wedge (a_3 \Leftrightarrow \mathtt{b_3})$
2	$(\mathtt{NF_a} \neq \mathtt{NF_b})$	$(\mathtt{a_1} \neq \mathtt{b_1}) \vee (\mathtt{a_2} \neq \mathtt{b_2}) \vee (\mathtt{a_3} \neq \mathtt{b_3})$	$(\mathtt{a_1} \oplus \mathtt{b_1}) \vee (\mathtt{a_2} \oplus \mathtt{b_2}) \vee (\mathtt{a_3} \oplus \mathtt{b_3})$
3	$(\mathtt{NF_a} > \mathtt{NF_b})$	$(\mathtt{a_1} < \mathtt{b_1}) \vee ((\mathtt{a_1} == \mathtt{b_1}) \wedge (\mathtt{a_2} > \mathtt{b_2})) \vee$ $((\mathtt{a_1} == \mathtt{b_1}) \wedge (\mathtt{a_2} == \mathtt{b_2}) \wedge (\mathtt{a_3} > \mathtt{b_3}))$	$(\neg \mathtt{a_1} \wedge \mathtt{b_1}) \vee ((\mathtt{a_1} \Leftrightarrow \mathtt{b_1}) \wedge (\mathtt{a_2} \wedge \neg \mathtt{b_2})) \vee$ $((\mathtt{a_1} \Leftrightarrow \mathtt{b_1}) \wedge (\mathtt{a_2} \Leftrightarrow \mathtt{b_2}) \wedge (\mathtt{a_3} \wedge \neg \mathtt{b_3}))$
4	$(\mathtt{NF}_a \geq \mathtt{NF_b})$	$(\mathtt{a_1} < \mathtt{b_1}) \vee ((\mathtt{a_1} == \mathtt{b_1}) \wedge (\mathtt{a_2} \geq \mathtt{b_2})) \vee$ $((\mathtt{a_1} == \mathtt{b_1}) \wedge (\mathtt{a_2} == \mathtt{b_2}) \wedge (\mathtt{a_3} \geq b_3))$	$(\neg \mathtt{a_1} \wedge \mathtt{b_1}) \vee ((\mathtt{a_1} \Leftrightarrow \mathtt{b_1}) \wedge (\mathtt{b_2} \Rightarrow \mathtt{a_2})) \vee$ $((\mathtt{a_1} \Leftrightarrow \mathtt{b_1}) \wedge (\mathtt{a_2} \Leftrightarrow \mathtt{b_2}) \wedge (\mathtt{b_3} \Rightarrow \mathtt{a_3}))$
5	$(\mathtt{NF_a} + \mathtt{NF_b})$	$\mathtt{S}_1^4 \equiv [\mathtt{C_1}, (\mathtt{a_1} \oplus \mathtt{b_1}) \oplus \mathtt{C_2},$ $(\mathtt{a_2} \oplus \mathtt{b_2}) \oplus \mathtt{C_3}, (\mathtt{a_3} \oplus \mathtt{b_3}) \oplus \mathtt{C_4}]$ $\mathtt{S}_1^3 \equiv (\mathtt{a}_i \wedge \mathtt{b}_i) \vee (\mathtt{C}_{i+1} \wedge (\mathtt{a}_i \oplus \mathtt{b}_i))$ $\mathtt{S_4} \equiv$ False	$[(\mathtt{a_1} \wedge \mathtt{b_1}) \vee (((\mathtt{a_2} \wedge \mathtt{b_2}) \vee ((\mathtt{a_3} \wedge \mathtt{b_3}) \wedge (\mathtt{a_2} \oplus$ $\mathtt{b_2}))) \oplus (\mathtt{a_1} \oplus \mathtt{b_1})),$ $(\mathtt{a_1} \oplus \mathtt{b_1}) \oplus ((\mathtt{a_2} \wedge \mathtt{b_2}) \vee ((\mathtt{a_3} \wedge \mathtt{b_3}) \wedge (\mathtt{a_2} \oplus \mathtt{b_2}))),$ $(\mathtt{a_2} \oplus \mathtt{b_2}) \oplus (\mathtt{a_3} \wedge \mathtt{b_3}),$ $(\mathtt{a_3} \oplus \mathtt{b_3})]$
6	$(\mathtt{NF_a} - \mathtt{NF_b})$	$\mathtt{S}_1^4 \equiv [\mathtt{C_1}, (\mathtt{a_1} \oplus \mathtt{b_1}) \oplus \mathtt{C_2},$ $(\mathtt{a_2} \oplus \mathtt{b_2}) \oplus \mathtt{C_3}, (\mathtt{a_3} \oplus \mathtt{b_3}) \oplus \mathtt{C_4}]$ $\mathtt{S}_1^3 \equiv (\mathtt{a}_i \wedge \mathtt{b}_i) \vee (\mathtt{C}_{i+1} \wedge (\mathtt{a}_i \oplus \mathtt{b}_i))$ $\mathtt{S_4} \equiv$ True	$[(\mathtt{a_1} \wedge \mathtt{b_1}) \vee (((\mathtt{a_2} \wedge \mathtt{b_2}) \vee ((\mathtt{a_3} \wedge \mathtt{b_3}) \vee (\mathtt{a_3} \oplus$ $\mathtt{b_3}) \wedge (\mathtt{a_2} \oplus \mathtt{b_2}))) \oplus (\mathtt{a_1} \oplus \mathtt{b_1})), (\mathtt{a_1} \oplus \mathtt{b_1}) \oplus ((\mathtt{a_2} \wedge$ $\mathtt{b_2}) \vee ((\mathtt{a_3} \wedge \mathtt{b_3}) \vee (\mathtt{a_3} \oplus \mathtt{b_3}))) \wedge (\mathtt{a_2} \oplus \mathtt{b_2}), (\mathtt{a_2} \oplus$ $\mathtt{b_2}) \oplus ((\mathtt{a_3} \wedge \mathtt{b_3}) \vee (\mathtt{a_3} \oplus \mathtt{b_3})), \neg(\mathtt{a_3} \oplus \mathtt{b_3})]$
7	$(\mathtt{NF_a} * \mathtt{NF_b})$	$\mathtt{M} \equiv \mathtt{NF_a} + \mathtt{NF_a} \ldots + \mathtt{NF_a}$ $\lvert\mathtt{NF_b}\rvert$ times $\mathtt{m_6} \equiv \mathtt{a_1} \oplus \mathtt{b_1}$	Replace additions by 5th row $\lvert\mathtt{NF_b}\rvert$ times
8	$(\mathtt{NF_a}/\mathtt{NF_b})$	$\lvert\mathtt{NF_a}\rvert - \lvert\mathtt{NF_b}\rvert - \lvert\mathtt{NF_b}\rvert \ldots - \lvert\mathtt{NF_b}\rvert$ $\mathtt{D} \equiv$ #times penultimate negative value $\mathtt{d_3} \equiv a_1 \oplus \mathtt{b_1}$	Replace subtractions by 6th row D times
9	$(\mathtt{NF}_a\%\mathtt{NF_b})$	$\lvert\mathtt{NF_a}\rvert - \lvert\mathtt{NF_b}\rvert - \lvert\mathtt{NF_b}\rvert \ldots - \lvert\mathtt{NF_b}\rvert$ $\mathtt{MOD} \equiv$ penultimate negative value $\mathtt{mod_3} \equiv$	Replace subtractions by 6th row D (8th row) times

range of [−4,3] (ie., n=3 bits) with bit-1 is the sign bit[4]:

$$a, b = \langle a_1, a_2, \ldots, a_n \rangle, \langle b_1, b_2, \ldots, b_n \rangle$$

where $a_i, b_i \in \{0, 1\}$; $1 \leq i \leq n$

Of course, we could have used larger widths in Table 1, but 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* ($\oplus$) (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 involves an implication in case of $\geq$ (row 4, col 3), or a disjunction of opposites in case of > (row 3, col 3).

Encoding arithmetic expressions is more complex. We use the term *n-signed bits* to mean an integer with n−1 bits for a value (ie., $0..2^{n-1}-1$), plus a sign bit. Addition and multiplication of bit-vectors can produce a result outside the domain range. For example, for 3 signed bits, if we perform '3+1', the result is '4', which requires 4 signed bits. The extra bit is the *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. The 6th operation is subtraction, which is a two's complement encoding of addition with an opposite sign bit (ie., $C_0 = True$). The multiplication pattern is row 7 of Table 1, which is a sign bit calculation plus a sequence of additions with a *double* bit-width. Division in row 8 is the times of the last but one subtraction of the second operand until the result is below zero. The modulo operation in row 9 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 PFs in CNF (Biere et al., 2009). Originally, Nemo applied the Tseitin's CNF transformation with Skolemization (Tseitin, 1983), the fastest known encoding to transform PFs into a CNF formula while maintaining model equivalence and model count (ie., not altering the total number of solutions). And now, Nemo2 supports other CNF encodings that theoretically can produce formulas that are faster to analyze at the cost of increased transformation times (Kuiter et al., 2023)

3. Nemo2

Manually applying bit-blasting to arithmetic requiring large bit-widths will take too much time. Therefore, we automated the process by developing Nemo2. Compared to the ICSR version Nemo (Munoz et al., 2022), this new version Nemo2 supports new input and output model formats from different state-of-the-art solvers. Additionally, Nemo2 now allows to extend/compose already modeled FMs with new NFs.

3.1. Tool overview

Fig. 1 presents an overview of Nemo2, in which a *modeling expert* defines a NFM for a given SPL. The new functionalities are tagged as such within red hexagons. Nemo2 provides a simple language to express 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;

[4] We updated addition and subtraction in Table 1 to an easier to debug alternative compared to the Munoz et al. (2022) version.

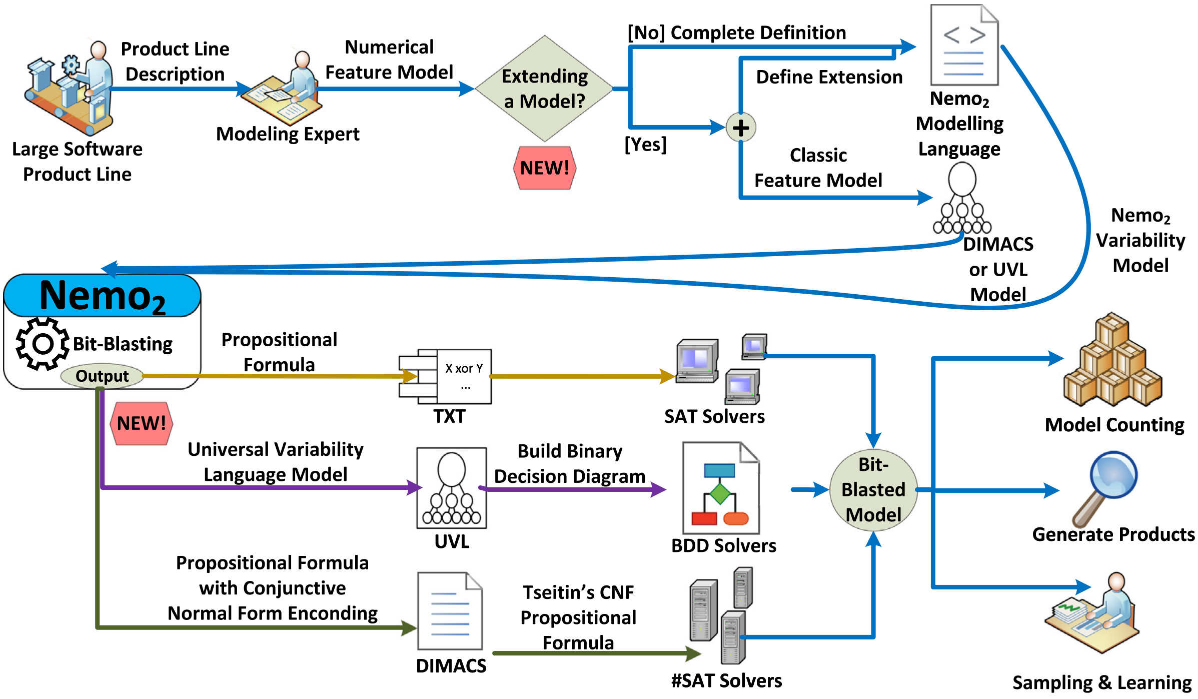

Fig. 1. Overview of the current version of the ℕ𝕖𝕞𝕠2 Tool.

- *Propositional Logic*: equivalences, implications, negations, conjunctions, disjunctions, parenthetical expressions, etc.;
- *Inequalities*: equal, not equal, greater (or equal), lower (or equal); and
- *Arithmetic*: addition, subtraction, multiplication, division, and modulo.

The default input to ℕ𝕖𝕞𝕠2 is a `.txt` file containing a complete ℕ𝔽𝕄 definition. Further, we can now extend an 𝔽𝕄 with new features and constraints by including them as a second file in one of the following formats:

- A `.txt` file if we are extending a ℙ𝔽.
- A `.uvl` file if we are extending a ℕ𝔽𝕄 in the *Universal Variability Language* (UVL) format.
- A `.dimacs` file if we are extending a ℕ𝔽𝕄 as a DIMACS.

For clarity purposes, we detail the default ℕ𝕖𝕞𝕠2 transformation procedure in Algorithm 1.

We also implemented three different output formats for the bit-blasted ℕ𝔽𝕄:

- A `.txt` file of an equivalent classical ℙ𝔽, a standard compatible with the state-of-the-art SAT solvers.
- A `.uvl` file of an equivalent UVL 𝔽𝕄, a standard compatible with the state-of-the-art #SAT solvers.
- A `.dimacs` file of an equivalent DIMACS CNF 𝔽𝕄, a standard compatible with the state-of-the-art BDD solvers.

As the grammar of each resulting format is different, a different set of state-of-the-art reasoners supports them. The main differences are:

- DIMACS is a set of clauses in CNF, while the classical ℙ𝔽 and the UVL model can contain implications, equivalences, nested clauses, and parenthesis.
- UVL model is a textual variability tree with a Root feature and the respective hierarchical constraints among features (ie., father and children) followed by independent cross-tree constraint clauses. On the other hand, the classical ℙ𝔽 is a single "all-in-one" clause that includes hierarchical constraints, and a Root variable is not needed.

Algorithm 1: ℕ𝕖𝕞𝕠2 Process (blue lines of Fig. 1)

```
Input: i) The NFM completely
defined in a .txt file
  ii) The output format: PF, UVL or DIMACS Parse features
  names;
Calculate features types;
Adjust NFs bit-widths;
Optimize data-types, domains, widths, and constraints;
Register the declared and calculated constraints;
Bit-blast the NFs and inequality equations;
Transform the bit-blasted NFM into a set of formulas;
if (PF or DIMACS) then
    Transform the formulas into a PF;
    if DIMACS then
        Transform the PF into its Tseitin CNF form;
        Transform the Tseitin CNF PF into DIMACS;
else if UVL then
    Transform the formulas into an UVL model;
Result: i) DIMACS, UVL or PF of the bit-blasted NFM
        ii) A .txt file matching NFs with bit-vectors
```

Consequently, a simple switch of the file extension will not make them cross-compatible.

In all of our three encodings, we identify each bit-vector with its original name plus the sequence of bits (Big Endian); In contrast, boolean features are identified as name plus *Boolean* keyword. We embed this information in the resulting bit-blasted ℕ𝔽𝕄, but it is purely informative and hence not considered by the automated reasoners. This information is placed as comments in the first lines of the classical ℙ𝔽 and the DIMACS files and alongside their occurrence in the hierarchy of the textual variability tree in the UVL model.

We now detail the structure of each supported format.

Table 2
DIMACS example for ‘‘A and (B or not C) and (B or D)’’.

Code	Description
c 1	variable A (variables first)
c 2	variable B
c 3	variable C
c 4	variable D
p cnf 4 3	header,CNF format, 4 variables, and 3 clauses
1 0	A (clauses last)
2 −3 0	and (B or not C)
2 4 0	and (B or D)

3.1.1. Bit-blasted ℕ𝔽𝕄 as a ℙ𝔽 in 𝕟𝕖𝕞𝕠𝟚

Mannion was the first to connect propositional formulas to product lines Mannion (2002). Time after, Batory formally defined the mapping between 𝔽𝕄s and ℙ𝔽s (Batory, 2005). In short, a ℙ𝔽 is a set of boolean variables and a propositional logic predicate that constrains the values of these variables. 𝕟𝕖𝕞𝕠𝟚's ℙ𝔽 output returns a single predicate nesting between parenthesis the different features by relating them with the standard And and Or connectives, the implication and equivalence operations, and the negation, in the forms of (,), &, |, =>, <=>, ! respectively. For example, Listing 1 presents 𝕟𝕖𝕞𝕠𝟚's ℙ𝔽 output for a bit-blasted ℕ𝔽𝕄 with one feature and two bounded ℕ𝔽s and the arithmetic constraint ‘‘C implies (A != B)’’ is:

```
# Vectorized Features:
#  v1 A_1
#  v2 A_2
#  v3 B_1
#  v4 B_2
#  C  Boolean
# Formula:
  !(!((v1 | !v2))) &
  !(!((v2 | v1)))  &
  !(!((v3 | !v4))) &
  (!(C) | !(!((v1 <=> v3 | !v2 <=> v4))))
```

Listing 1: 𝕟𝕖𝕞𝕠𝟚 ℙ𝔽 output for: A ∈ [-1,0]; B ∈ [-1,1]; C Boolean in "C requires (A != B)"

As shown by gold arrows in Fig. 1, the generated ℙ𝔽 is supported by SAT solvers to create products or enumerate configurations, useful for fast probabilistic sampling and learning (Heradio et al., 2022).

3.1.2. Bit-blasted ℕ𝔽𝕄 as an UVL 𝔽𝕄 in 𝕟𝕖𝕞𝕠𝟚

The first formal proposal and partial definition of the UVL was published by Sundermann et al. (2021s). Its adoption is rapidly increasing, with tool support by state-of-the-art modeling and reasoning tools such as Feature IDE (Sundermann et al., 2021b) and Pure::variants (Romano et al., 2022).

The main idea behind UVL is to be a common input between the different variability modeling tools currently in use. Additionally, it incorporates most of the modeling requirements from the SPL community (Berger and Collet, 2019), such as being human-friendly and having a soft learning curve. While the current UVL version covers 𝔽𝕄s with feature-wise attributes, compatible reasoning tools support only classical 𝔽𝕄s (Sundermann et al., 2021b). Nevertheless, this includes all sorts of cardinality and related definitions and abstract features that can be referenced multiple times (ie., clones). Its textual representation, similar to an 𝔽𝕄 tree graph, follows an indented approach with primary keywords that divide each section such as features, constraints, import, etc. Listing 2 presents 𝕟𝕖𝕞𝕠𝟚's UVL output for a bit-blasted ℕ𝔽𝕄 with the same example of one feature and two bounded ℕ𝔽s and the arithmetic constraint ‘‘C implies (A != B)’’.

```
features
  Root
    optional
      v1 # A_1
      v2 # A_2
      v3 # B_1
      v4 # B_2
      C  # Boolean
constraints
  !(!((v1 | !v2)))
  !(!((v2 | v1)))
  !(!((v3 | !v4)))
  (!(C) | !(!((v1 <=> v3 | !v2 <=> v4))))
```

Listing 2: 𝕟𝕖𝕞𝕠𝟚 UVL output for: A ∈ [-1,0]; B ∈ [-1,1]; C Boolean in "C requires (A != B)"

As shown with the purple arrows in Fig. 1, we can use the resulting UVL file to generate products or count configurations with state-of-the-art BDD solvers (Heradio et al., 2022).

3.1.3. Bit-blasted ℕ𝔽𝕄 as a *DIMACS CNF* in 𝕟𝕖𝕞𝕠𝟚

DIMACS dates back to 1993 and is the de-facto input format standard for SAT solvers.[5] 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 Tseitin-CNF ℙ𝔽 format. 0 is a reserved keyword for a clause delimiter. DIMACS format identifies features sequentially with a unique numerical index. Table 2 illustrates a DIMACS file:

In this output case, we need to consider that a CNF Tseitin transformation of a bit-blasted ℕ𝔽𝕄 generates extra variables. Table 3 continues with a bit-blasted example in DIMACS of a bit-blasted ℕ𝔽𝕄 with one feature and two bounded ℕ𝔽s and the arithmetic constraint ‘‘C implies (A != B)’’. As shown with the green arrows in Fig. 1, the generated DIMACS file can be used to count configurations with a #SAT solver efficiently.

3.2. Numerical Feature Modeling in 𝕟𝕖𝕞𝕠𝟚

Most feature modeling languages today are tool-specific (Raatikainen et al., 2019), eg., Clafer (Bąk et al., 2010). For 𝕟𝕖𝕞𝕠𝟚, we abstract ℕ𝔽𝕄s to only two entities (Munoz et al., 2021; Horcas et al., 2020): generic variables and constraints. Our motivation was to reduce 𝕟𝕖𝕞𝕠𝟚's learning curve. Consequently, we present the cheat sheet in Table 4.

Listing 3 illustrates most of the types of supported clauses:

```
def A bool 0     # 0 means new feature
def B bool B     # named in adjunt FM as B
def C bool 0
def D_unsigned [0:1]
def E_unsigned [0:3]
def F_signed [-1:1]
def G_enum_signed [-9, -3, 0, 3]
def H_constant [-2]
ct  C -> B
ct  A -> (G = 0)
ct  A or B
ct (G_enum_signed*H_constant) ≤ E_unsigned
```

Listing 3: Example of extending with 𝕟𝕖𝕞𝕠𝟚 an 𝔽𝕄 with new boolean and numerical features and constraints

Sequentially, Listing 3 keywords mean:

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

Table 3
Nemo2 DIMACS output for: A ∈ [−1,0]; B ∈ [−1,1]; C Boolean in‘‘C requires (A != B)’’.

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	Tseitin4
c 9	Tseitin5
c 10	Tseitin6
c 11	C Boolean
p cnf 11 24	header, cnf format, 11 variables, and 24 clauses
−1 5 0	(not Abit1 or not Tseitin1)
2 5 0	and (Abit2 or Tseitin1)
1 −2 -5 0	and (Abit1 or not Abit2 or not Tseitin1)
5 0	and Tseitin1
−2 6 0	and (not Abit2 or Tseitin2)
1 6 0	and (Abit1 or Tseitin2)
2 −1 -6 0	and (Abit2 or not Abit1 or not Tseitin2)
6 0	and Tseitin2
7 −3 0	and (Tseitin3 or not Bbit1)
4 7 0	and (Bbit2 or Tseitin3)
3 −4 -7 0	and (Bbit1 or not Bbit1 or not Tseitin3)
7 0	and Tseitin3
−2 -4 −8 0	and (not Abit2 or not Bbit2 or not Tseitin4)
2 4 −8 0	and (Abit2 or Bbit2 or not Tseitin4)
2 −4 8 0	and (Abit2 or not Bbit2 or Tseitin4)
−2 4 8 0	and (not Abit2 or Bbit2 or Tseitin4)
−1 -3 −9 0	and (not Abit1 or not Bbit1 or not Tseitin5)
1 3 −9 0	and (Abit1 or Bbit1 or not Tseitin5)
1 −3 9 0	and (Abit1 or not Bbit1 or Tseitin5)
−1 3 9 0	and (not Abit1 or Bbit1 or Tseitin5)
−9 10 0	and (not Tseitin5 or Tseitin6)
−8 10 0	and (not Tseitin4 or Tseitin6)
8 9 −10 0	and (Tseitin4 or Tseitin5 or not Tseitin6)
−11 10 0	and (not C or Tseitin6)

Table 4
Cheat Sheet of NFM Modeling with Nemo2.

Keyword	Description
def Name Domain	Defines a feature by its name and domain (eg., range of values)
[X]	Indicates a NF with a constant value X
[X:Y]	Indicates a range between X and Y inclusive
[X,Y,Z]	Indicates an enumerated type with values X, Y or Z
ct	Indicates the start of a definition of a single constraint
and/or	Are conjunctions and disjunctions
<->/->/neg	Are equivalences, implications and negations
=/>/</>=/<=/!=	Are the equalities/inequalities
+/−/*///%	Are the numerical operators

1. A bool and C bool 0: boolean features, newly defined as tagged by zero (0) identifier. This is necessary if we are extending or composing models, 0 means that they have not been included in the FM tree yet;
2. B_bool B: a boolean feature defined in the attached FM that we are extending where B is its exact name in that model. In Fig. 2 we graphically summarize the inputs that are needed for this concrete extension of a previously modeled FM;
3. D_unsigned: a natural NF with inclusive values 0 to 1 in two's complement encoding;
4. E_unsigned: another natural NF with inclusive values 0 to 3 in two's complement encoding;

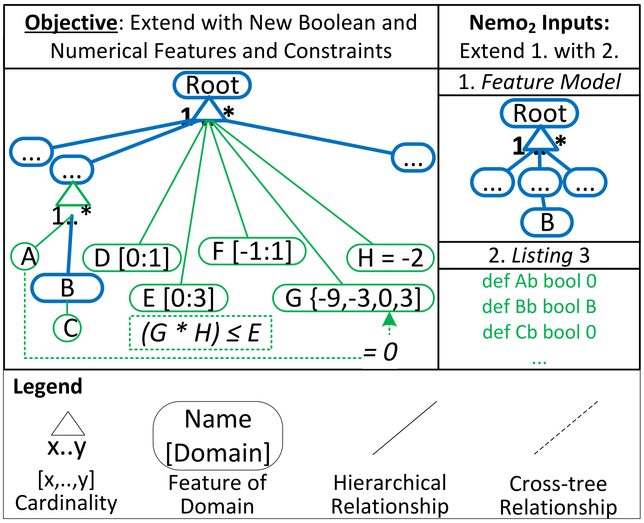

Fig. 2. Extending with Nemo2 an FM with Listing 3 NFM.

5. F_signed: an integer NF with inclusive values −1 to 1 in two's complement encoding;
6. G_enum_signed: an enumerated integer NF with exactly 4 values in two's complement encoding;
7. H_constant: a constant integer NF with a value of −2;
8. C -> B: A propositional logic requirement;
9. A -> (G = 0): A propositional logic requirement with an arithmetic equality;
10. A or B: A propositional logic disjunction;
11. (G_enum_signed ∗ H_constant) ≤ E_unsigned: An arithmetic constraint.

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. As a formal definition, we present in Listing 4 the complete Nemo2's modeling context-free grammar in an extended Backus–Naur form notation:

```
NumericalFeatureModel = Features? Constraints?

Features = (BoolFeature | NumFeature)+\n
<BoolFeature> = FeatureSpec <'bool'> Name
NumFeature = FeatureSpec
 <'['>(Number|Range|Enumeration)<']'>
<FeatureSpec> = <'def'> Name
Range = (Number? <':'> Number?)
<Enum> = (Number<','?>)+

Constraints = <'ct'> (Formula)+\n
<Formula> = Predicate|Equation
Predicate = <'('>?BoolFormula|IneqEquation<')'>?
 (Connective<'('>BoolFormula|IneqEquation<')'>)*
<BoolFormula> = not? BoolFeature
 (Connective not? BoolFeature)*
Connective = <'and'>|<'or'>|<'->'>|<'=>'>
<IneqEquation> = <'('>?NumEquation<')'>?
 (Ineq <'('>NumEquation<')'>)*
Ineq = <'='>|<'>'>|<'<'>|<'>='>|<'<='>|<'!='>
<NumEquation> = NumFeature (Arith NumFeature)*
Arith = <'+'>|<'-'>|<'*'>|<'/'>|<'%'>

<Name> = #'[a-zA-Z_0-9]+'
<Number> = #'[-]?[0-9]+'
```

Listing 4: Nemo2's context-free grammar in a Backus–Naur form

3.3. Automatic calculation of minimal bit-widths

Nemo2 is a cross-platform tool developed in Python 3.11.0 x86_64. It posed several engineering challenges. First, Nemo2 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).[6] If any value of a NF is negative, it is considered an `integer`.

Second, Nemo2 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 example, the 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, Nemo2 readjusts the previous computed widths based on NFM constraints. Leaving aside boolean features, every NF involved in operations with another NFs must have the same type and bit-width in order to apply bit-blasting. 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 example, transforming a `natural` into an `integer` NF adds one bit for the sign.

Fourth, Nemo2 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 $2^{bit-width} - 1$.
- **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 $bit - width_{updated} = (bit - width_{current} \times 3) + 1$.

3.4. Nemo2 *Optimizations by Pre-Processing the* NFM

Bit-blasting and Tseitin transformations create different size CNF PFs depending on the equation. Nemo2 pre-process the input NFM to reduce or replace the NFs domains and arithmetic constraints to produce smaller bit-blasted models. This not only reduces the number of lines of the resulting file, but also the number of features and the size of the clauses of the resulting model independently of the output. This causes a reduction of the feature space size and complexity, and consequently the performance and scalability of automated reasoning like decreasing counting time (Shih and Cheng, 2005).

First of all, Nemo2 removes duplicated constraints when possible. For example, in case of the constraints `A<1` and `A<2` the first one is redundant. Additionally, Nemo2 dynamically prioritizes `natural` NFs, as unsigned operations need smaller bit-widths and produce smaller PFs due to removing sign-bits. For example, if 1 `integer` and 9 `natural` NFs are present in a `integer` addition operation, we need 10 sign-bits, as the operation and all of its operands must have a compatible domain (i.e., `integer`). In the case of the DIMACS output we also consider which operations are creating more Tseitin artificial features, and hence replace those ones by their more optimal alternatives. Concretely:

1. >/</+/− do not create extra variables;
2. ≥/≤ create (*bit-width*−1) Tseitin variables in the NFs involved;
3. = creates (*bit-width*) Tseitin variables in the NFs involved;
4. =/ 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 {≥,≤} by {>,<} respectively. (eg., `A≥1` and `A≤2` are equivalent to `A>0` and `A<3`).

4. Evaluation

The following research questions evaluate Nemo2, including the complete set of arithmetic and its three different transformations Tseitin CNF DIMACS, classical PF and UVL model as detailed in Section 3.

RQ1: How do the three different transformations of Nemo2 scale for different bit-widths and constraints?
RQ2: How do the three different transformations of Nemo2 scale for real-world NFMs?
RQ3: How well do bit-blasted NFMs generated by Nemo2 perform on model counting with the state-of-the-art solvers for different arithmetic constraints?
RQ4: How well do bit-blasted NFMs generated by Nemo2 perform on random sampling with the state-of-the-art solvers for real-world NFMs?

In short, **RQ1-2** evaluate Nemo2's scalability on transforming synthetic and real-world models, and **RQ3-4** evaluate the scalability of the state-of-the-art solvers on counting and sampling those outputs (ie., bit-blasted NFMs). 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 22.04 LTS X86_64.

RQ1: How do the three different transformations of Nemo2 scale for different bit-widths and constraints?
In this RQ, we evaluate Nemo2's general runtime when transforming all sorts of NFs with boolean and arithmetic constraints. All tests are performed for comparison purposes for the three outputs detailed in Section 3: PF, UVL and Tseitin CNF DIMACS. We start by transforming the most complex types of NFM operations, ie., 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 analyzed the first set of 5 NFM constraints defined by:

1. `(A + B) = C`
2. `(A - B) = C`
3. `(A * B) = C`
4. `(A / B) = C`
5. `(A % B) = C`

Formulas with different bit widths (#*b*) from 2 up to 16 bits step 2 were generated. Remember that the operations that create more carry-bits produce the maximum bit-width.

Fig. 3 shows the first set of results. Regarding the different outputs, we can visualize that the Tseitin CNF DIMACS transformation is the slowest, closely followed by UVL and then PF. The explanation is calculating a PF is the final step to generate its respective file or when generating the UVL model. Still, it is an intermediate step when computing any CNF formula.

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

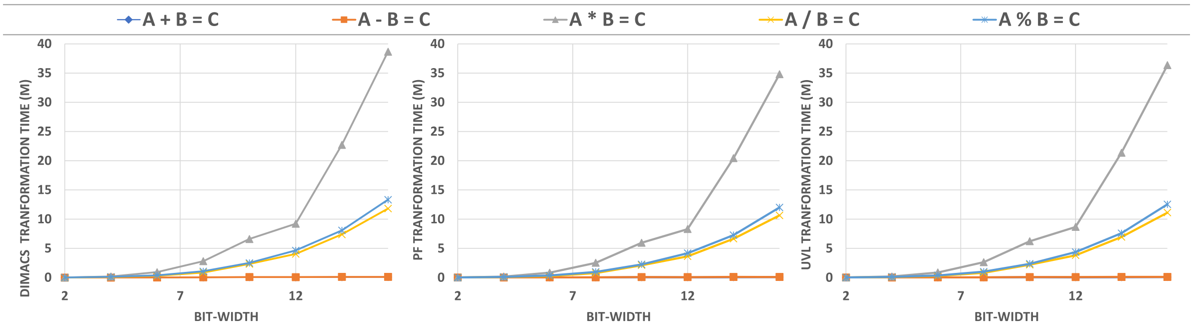

Fig. 3. Nemo2 runtime in seconds of arithmetic operations and equations sets.

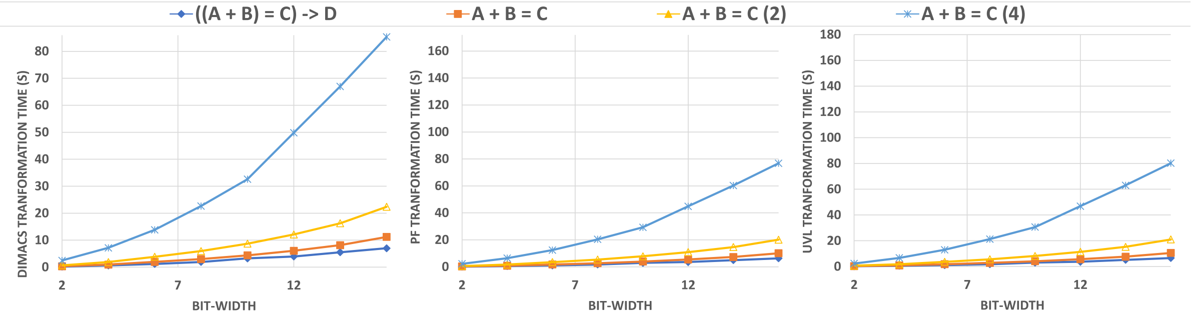

Fig. 4. Nemo2 runtime in seconds of arithmetic operations and equations sets.

Regarding constraints, Nemo finishes *instantly* for addition and subtraction operations. However, the runtime is slightly exponential for division and modulo and truly exponential for multiplication due to the carry bits of the operations. Nevertheless, all 16 bit-width transformations finished in under 40 min. A possible solution to reduce the transformation time of multiplication operations is to discretize the possible values of the NFs. A common solution would be considering only the pair or odd values, which will decrease the bit-width needed by half.

As the number of NF variables is proportional to the bit-width, the Tseitin's transformation guarantees a linear increase O(3n+1) (Tseitin, 1983). Hence, they cannot be the reason behind the scalability differences between the operations. The issue comes from the carry-bits, as 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.

Further, to evaluate all types of constraints' size and casuistic and test their performance behavior, we analyze logic and arithmetic mixed nested constraints and up to four conjuncted numerical constraints. Following the previous procedure, we prioritize the less demanding operations (ie., =, +, ⇒) to reduce interactions for more precise 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. 4 shows the second set of results. Regarding the different outputs, they are similar to those of Fig. 3, meaning that the Tseitin CNF DIMACS transformation is the slowest due to needing extra computational steps. Regarding nested and stacked constraints, processing all equalities takes a maximum of 85 s.

Conclusion: ***Nemo2 NFs are unbounded by default, but their encoding scales up to 16 bit-width per number with transformation times of approximately one minute. The exception is multiplication, which takes 40 min to bit-blast for a 16 bit-width. Mixing, nesting, and conjuncting operations produce a linear increase in transformation time. Additionally, there are no big differences between the different output formats, the classical PF being the fastest transformation.***

RQ2: How do the three different transformations of Nemo2 scale for real-world NFMs?

This RQ evaluates Nemo2's specific runtime when transforming large real-world NFMs. Again, all tests are performed for the PF, UVL and Tseitin CNF DIMACS transformations.

We evaluate a total of 12 real-world NFMs. We obtained Dune, HSMGP, HiPAcc, and Trimesh from (Oh et al., 2019); MOTIV from (Galindo et al., 2014); axTLS, Fiasco, and uClibc-ng from (Siegmund et al., 2015); and Busybox 1.18.5, Busybox 1.28 and Linux 2.6.33.3 from (Sundermann and Feichtinger, 2021) and extended with their respective NFs and constraints defined in Catenazzi (2022). When the NF domain was not properly defined, we restricted them to the minimum necessary based on its contextual definition; for example, restricting a NF delay in seconds to a maximum of 7 days instead of years (Catenazzi, 2022). Table 5 lists and summarizes these NFMs, where each system has a different number of NFs and/or different configuration space size. They are ordered first by source and then by space size, reaching up to NFM with a space size of approximately 5.66×10^{1953}.[7] All the input NFMs and the three Nemo2 different output formats for each NFM are uploaded to GitHub and Zenodo. [8]

As we can visualize in Table 6, the larger the NFM, the more time that Nemo2 needs to bit-blast it. Nevertheless, the scalability is linear, as most models are transformed in less than 10 s, and a colossal one, like Linux, takes approximately 45 min. Regarding

[7] Linux 2.6.33.3 NFM space size of 5.66×10^{1953} is estimated with the tool from Horcas et al. (2022b) due to not existing tools that can yet accurately count such colossal FMs

[8] Nemo2 data-set can be downloaded from:
- https://github.com/danieljmg/Nemo2_tool
- https://doi.org/10.5281/zenodo.7781025

Table 5
List of the Real-World Numerical Feature Models analyzed in RQ2 and RQ4.

Source	ℕ𝔽𝕄	Description	#F	#ℕ𝔽s	#Configs
`FSE2015`(Siegmund et al., 2015)	Dune	Multi-grid solver	11	3	2,304
	HSMGP	Stencil-grid solver	14	3	3,456
	HiPAcc	Image processing	33	2	13,485
	Trimesh	Triangle mesh library	13	4	239,360
`ISSTA14` (Galindo et al., 2014)	MOTIV	Mobile Video Sequence	8	13	3.52×10^{30}
`UMA18` (Horcas, 2018)	WeaFQAs	Quality Attributes Weaver	240	5	1.38×10^{40}
`KConfig`(Foundation, 2018; Sundermann and Feichtinger, 2021)	Fiasco	Real-time microkernel	234	5	3.06×10^{12}
	axTLS	Client–server library	94	9	4.96×10^{38}
	uClibc-ng	C Language library	269	6	8.20×10^{45}
	Busybox 1.18.5	Embedded Linux	631	12	1.34×10^{191}
	Busybox 1.28	Embedded Linux	1100	12	1.53×10^{248}
	Linux 2.6.33.3	Operating System Kernel	6467	55	$\sim 5.66 \times 10^{1953}$

Table 6
Nemo2's runtime in seconds when bit-blasting real-world ℕ𝔽𝕄s in three different formats (ie., Tseitin CNF `DIMACS`, classic ℙ𝔽 and UVL model).

seconds	Dune	HSMGP	HiPAcc	Trimesh	MOTIV	WeaFQAs	Fiasco	axTLS	uClibc-ng	Busybox 1.1	Busybox 1.2	Linux 2.6
DIMACS	4.77	4.25	7.01	10.33	9.06	8.99	8.67	97	190	589	598	2713
ℙ𝔽	4.23	3.88	5.45	8.76	9	8.21	7.79	86	169	489	505	2108
UVL	4.53	4	6.77	9.79	9.03	8.63	8.08	93	178	547	555	2545

Table 7
Counting time in seconds of synthetic ℕ𝔽𝕄s of a bit-width of 12 transformed with Nemo2.

Counting Time (bit-width 12)	Glucose3	sharpSAT	Flamapy BDD	BDDSampler
(A + B) = C	Time-out	0.1 s	Time-out	1.17 s
(A - B) = C	Time-out	0.1 s	Time-out	1.17 s
(A ∗ B) = C	Time-out	0.7 s	Time-out	4.49 s
(A / B) = C	Time-out	32.15 s	Time-out	8.88 s
(A % B) = C	Time-out	51.85 s	Time-out	8.79 s
((A + B) = C) ⇒ D	Time-out	26.1 s	Time-out	2.1 s
(A + B) = C (2)	Time-out	16.35 s	Time-out	2.3 s
(A + B) = C (4)	Time-out	37.22 s	Time-out	4 s

Table 8
Sampling time of Nemo2's synthetic ℕ𝔽𝕄s with bit-width 12.

Sampling time (bit-width 12)	**Flamapy BDD**	**BDDSampler**
(A + B) = C	Time-out	6.56 s
(A - B) = C	Time-out	6.54 s
(A ∗ B) = C	Time-out	34.05 s
(A / B) = C	Time-out	98.06 s
(A % B) = C	Time-out	96 s
((A + B) = C) ⇒ D	Time-out	12.06 s
(A + B) = C (2)	Time-out	12.01 s
(A + B) = C (4)	Time-out	22.22 s

the differences between the three output formats, again, the classical ℙ𝔽 is the fastest, and UVL and Tseitin CNF `DIMACS` increase those times by an average of 10% and 15% respectively. As we can see by comparing with **RQ1**, **RQ2** runtimes seem much more scalable. The reason is that most of the ℕ𝔽 constraints present in real-world SPLs are numerical inequalities, or if arithmetic is involved, the bit-width tends to be small.

***Conclusion:* Nemo2 *linearly scales when transforming large real-world* ℕ𝔽𝕄s *in any of its three output formats, as its runtimes are kept below 45 min even for a colossal* ℕ𝔽𝕄s. UVL *and Tseitin* CNF DIMACS *transformations takes 10% and 15% more time than the equivalent* ℙ𝔽.**

RQ3: How well bit-blasting ℕ𝔽𝕄s generated by Nemo2 perform when model counting with the state-of-the-art solvers for different arithmetic constraints?

In this RQ, we evaluate the performance of the same arithmetic constraints of **RQ1** when model counting and uniform random sampling with different state-of-the-art tools. In our previous publication (Munoz et al., 2022), we selected three automated solvers, each of them from another type – `sharpSAT` (Thurley, 2006) as a #SAT solver, Z3[9] as an SMT solver, and Clafer[10] as a CP solver. However, Z3 and clafer did not properly scale, as counting could take hours in those solvers compared to 0.1 s in `sharpSAT`. Part of the reason that we discovered is that solvers that natively support ℕ𝔽s are currently less polished and hence less efficient reasoners. Therefore, for this work, we replaced Clafer and Z3 with three additional solvers: Glucose3 (Audemard and Simon, 2018) as an SAT solver, and Flamapy BDD (Horcas et al., 2022a) and BDDSampler (Heradio et al., 2022) BDD solvers.

Those three solvers are all integrated into the Flamapy tool. FLAMA acts as a format proxy by providing ℙ𝔽 and UVL model input support to those three solvers. Hence, we do not need to generate tool-specific models for those solvers. With them, alongside `sharpSAT`, we can perform an efficient model counting, and with the BDDs we can achieve efficient uniform random sampling. From now on, if the counting or sampling surpasses 72 hours, we consider it a time-out due to a high probability of never finishing.

Counting results are presented in Table 7. To be consistent with **RQ1** conclusions, the ℕ𝔽s are restricted to bit-widths of 12 (ie., inclusive range [−2048, 2047]). As we can see, half of the state-of-the-art tools cannot count ℕ𝔽𝕄s with complex arithmetic and nested operations for bitou-widths of 12. On the other hand, while multiplication was the most costly to bit-blast by Nemo2 as seen in **RQ1**, it is not by far the most complex to count. On the other hand, division and specially modulo are very slow to count compared to the rest of the operations, with increments of 320% and 530%, respectively. Additionally, nesting increments counting a 260%, which is almost the double of duplicating constraints. In general, adding new constraints does not create n-wise influences on performance. Regarding the differences between the solvers, Glucose and Flamapy BDD time-out in all cases. On the other hand, while `sharpSat` tends to be 100% faster than BDDSampler for lower complexities, it is the opposite for the more complex ones. Consequently, BDDSampler scalability is higher than that of `sharpSat`.

[9] Z3py: https://github.com/Z3Prover/z3

[10] Clafer: https://www.clafer.org/

Table 9
Model counting time of synthetic ℕ𝔽𝕄s transformed with ℕ𝕖𝕞𝕠𝟚.

Counting time	**Glucose3**	sharpSAT	**Flamapy BDD**	**BDDSampler**
Dune	0.37 s	0.01 s	0.03 s	0.44 s
HSMGP	69.43 s	0.01 s	0.03 s	0.51 s
HiPAcc	37.08 s	0.01 s	0.05 s	0.6 s
Trimesh	180.98 s	0.01 s	0.05 s	0.71 s
MOTIV	Time-out	0.01 s	Time-out	3.15 s
WeaFQAs	Time-out	0.01 s	Time-out	2.9 s
Fiasco	Time-out	0.01 s	Time-out	1.11 s
axTLS	Time-out	0.01 s	Time-out	2.45 s
uClibc-ng	Time-out	0.01 s	Time-out	4.95 s
Busybox 1.1	Time-out	4.3 h	Time-out	Time-out
Busybox 1.2	Time-out	5 h	Time-out	Time-out
Linux 2.6	Time-out	Time-out	Time-out	Time-out

Table 10
Sampling time of real-world ℕ𝔽𝕄s transformed with ℕ𝕖𝕞𝕠𝟚.

Sampling time	**Flamapy BDD**	**BDDSampler**
Dune	2.79 s	3 s
HSMGP	2.41 s	3 s
HiPAcc	5.5 s	3 s
Trimesh	5.57 s	3.1 s
MOTIV	Time-out	4.61 s
WeaFQAs	Time-out	3.68 s
Fiasco	Time-out	3.2 s
axTLS	Time-out	8.13 s
uClibc-ng	Time-out	9.73 s
Busybox 1.1	Time-out	Time-out
Busybox 1.2	Time-out	Time-out
Linux 2.6	Time-out	Time-out

Sampling results are presented in Table 8. Sample sizes are dynamically calculated by the Slovin's formula:

$$\#Samples = \frac{Size}{(1 + Size * Error^2)}$$

Again, Flamapy BDD time-out for all cases and the trends discussed for each operation in counting are similar for sampling. Concretely, division and modulo are the slowest and equally slow, and nesting is equally complex than adding new constraints.

Conclusion:* ℕ𝔽𝕄s *with complex constraints and large bit-widths bit-blasted with* ℕ𝕖𝕞𝕠𝟚 *are compatible with state-of-the-art solvers for counting and sampling. Unfortunately, the complexity that arithmetic adds is a time-out for Glucose3 and Flamapy BDD. Regarding counting,* sharpSat *is 100% faster than BDD Sampler for less complex operations like addition and multiplication, being the opposite for more complex operations like division, modulo, and nesting. Regarding sampling, BDDSampler performed all the constraints between 6 and under 98 s, with times increasing proportionally to the complexity of the operation. Finally, there is no relationship between the computational cost of bit-blasting a* ℕ𝔽𝕄 *and the analyses of those models

RQ4: How well bit-blasting ℕ𝔽𝕄s generated by ℕ𝕖𝕞𝕠𝟚 perform when random sampling with the state-of-the-art solvers for real-world ℕ𝔽𝕄s?

In this RQ, we perform the same reasoning operations that in **RQ3** but for the real-world ℕ𝔽𝕄s of Table 5. The number of samples and time-outs are consistent with **RQ3**.

The model counting results are shown in Table 9. When visualizing them, we can obtain several conclusions. The size of the ℕ𝔽𝕄 affects the tool's scalability, but the number and complexity of the ℕ𝔽s can greatly affect too. An example of this is the counting times of WeaFQAs and MOTIV — while WeaFQAs has a considerably larger space size, it is faster to count than MOTIV, which has more ℕ𝔽𝕄s. Additionally, we can conclude that regular SAT solvers like Glucose3 should not be considered for ℕ𝔽𝕄 analyses. BBD solvers are very fast, but there is a point where the construction of the BDD requires so many resources (eg., memory RAM) that the system crashes. On the other hand, sharpSat was generally the fastest and, most importantly, could count even colossal space sizes with a special mention to Busybox 1.2 in just 5 h. However, it is necessary to mention that we needed to increase the virtual memory of our testing computer to 1 Terabyte, as otherwise, the system would crash. Presented time-outs are not reduced by increasing the virtual memory in the rest of the solvers, and likewise if increasing it further than 1 Terabyte for sharpSat.

The uniform random sampling results are shown in Table 10. In this case, colossal spaces such as Busybox and Linux time-out in all contexts. Flamapy can neither sample large ℕ𝔽𝕄s, but tends to be a bit faster than BDDSampler. Nevertheless, BDDSampler scaled up to the colossal SPL uClibc-ng in under 10 s. If we did not apply the pre-processing of Section 3.4, we can expect an increase in the number of time-outs. Approximate counting and sampling techniques could reduce the time-outs, but those are configuration space reasoning techniques that are out of the scope of this work.

Conclusion: large real-world* ℕ𝔽𝕄s *bit-blasted with* ℕ𝕖𝕞𝕠𝟚 *are compatible with state-of-the-art solvers for counting and sampling. texttttsharpSat is the best performant solver for counting, with an analysis time of 0.01 s for many* ℕ𝔽𝕄s*, and just 5 h for Busybox 1.2* SPL*. Regarding sampling, BDDSampler presents runtimes between 3 and 10 s, scaling up to space sizes of* 8.20×10^{45} *(ie., uClibc-ng).

5. 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 (Systems, 2012). For **RQ3-4**, we used the counting methods and default options that the developers of each solver propose. We prepared a variety of synthetic constraints to test the limitations of ℕ𝕖𝕞𝕠𝟚 and, respectively, its outputs in the state-of-the-art solvers. Enumerated ℕ𝔽 domains with very distant values were encoded as the minimal number of alternatives in a single ℕ𝔽 to reduce the performance noise that those kinds of domains could create. For example, an integer ranger of just 4 enumerated values "[0, 1, 10, 512]" is defined as a bit-vector of width 3 instead of directly with width 10. This reduces the resulting bit-blasted 𝔽𝕄 size and complexity, meaning fewer bits and Tseitin features.

External validity. We used the 12 real-world SPLs of Table 5, which have different numbers of features, domains, constraints, and space sizes, including colossal ℕ𝔽𝕄s. For complex constraints, we evaluated synthetic models. While we are aware that our results may not generalize to all SPLs, their trends are identical in different cases. Similarly, although currently state-of-the-art, the selected solvers could be superseded by faster alternatives in the future. Additionally, a manual bit-blasting approach for ℕ𝔽s and basic operations was successfully applied for counting-based optimizations of SPLs (Munoz et al., 2019a). This was extended for the complete arithmetic set and automated with ℕ𝕖𝕞𝕠 in Munoz et al. (2022). ℕ𝕖𝕞𝕠𝟚 extends that version by supporting new input and output model formats from different state-of-the-art solvers: Tseitin CNF DIMACS, regular ℙ𝔽s and UVL models. This allows state-of-the-art solvers for classical 𝔽𝕄s like BDD algorithms to support ℕ𝔽𝕄s. Additionally, ℕ𝕖𝕞𝕠𝟚 now allows to extend/compose already modeled 𝔽𝕄s with new ℕ𝔽𝕄s.

6. Related work

Work tackling ℕ𝔽𝕄s is rare (Marchezan et al., 2022). Some considered ℕ𝔽s as classical features with just present/absent states (Berger et al., 2013; Oh et al., 2019; Döller and Karagiannis, 2021). Some encoded ℕ𝔽s as alternative features, where each value of a ℕ𝔽 was considered a distinct feature (Kästner et al., 2011). Shi (Shi, 2017) used a single type of feature called 'pseudo-boolean' with only Successor (+1) and Predecessor (−1) operations. In Benavides et al. (2010), each boolean feature had related attributes – a set of variables in the form ⟨name, value, domain⟩. However, attributes and ℕ𝔽s are essentially different: attributes are not nodes of the variability tree, and as opposed to a ℕ𝔽, a change in the value of an attribute does not result in a different configuration (Munoz et al., 2018). Hence, counting the size of a product space will return a lower-than-expected value.

SMT and CP solvers natively support the representation and reasoning of ℕ𝔽𝕄s. 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 (Phan, 2015), being unaware of allocated memory or domain definitions (eg., undefined maximum of x in $x \geq 1$). In SAT theory, all variables are bounded (ie., boolean). Consequently, SMT approximation counting has been proposed (Chistikov et al., 2017). STP solver (Ganesh and Dill, 2006) implements a bit-vector approach for counting. It performs array optimizations, arithmetic, and Boolean simplifications before bit-blasting to MiniSat Sorensson and Een (2005). While it works to test satisfiability by counting at least one, it does not preserve counting or model equivalence. This aligns with the most recent model counting competition (2020), where they tested 34 versions of the 8 fastest counting solvers. Model counting is more commonly found in *Binary Decision Diagrams* (Bryant, 2018) and SAT-based (Thurley, 2006) solvers. The results indicate that while fast, even so-called 'exact solvers' count a close but inexact number of configurations.

Simplification of ℕ𝔽𝕄s usually reduces reasoning time. However, those beyond the ones implemented in ℕ𝕖𝕞𝕠 do not preserve counting or model equivalence (Chakraborty et al., 2021). Nevertheless, the bit-width bottleneck is shared even in solutions that perform approximate counting. An example is Boolector reasoner (Brummayer and Biere, 2009), which lazily instantiates array axioms and macros. Even Z3 (Moura and Bjørner, 2008) applies bit-blasting to every operation besides equality, which is then handled by specific algorithms.

7. 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 more important to understand and to provide support to. Key reasoning operations are model counting and sampling. Unfortunately, while automated solvers can analyze 𝔽𝕄s, they were not developed with the objective of counting or sampling ℕ𝔽𝕄s. Again, counting configurations is key to finding near-optimal SPL configurations (eg., find one of the top configurations minimizing the run-time of a given benchmark (Munoz et al., 2019a; Oh et al., 2017; Heradio et al., 2022)).

We developed ℕ𝕖𝕞𝕠𝟚, a prototype that automatically optimally pre-process and transforms ℕ𝔽𝕄s to three different formats: a Tseitin CNF DIMACS file, a classical ℙ𝔽, and a UVL model. ℕ𝕖𝕞𝕠𝟚 represents ℕ𝔽s as bit-vectors through bit-blasting, while arithmetic constraints are encoded as propositional clauses. We evaluated ℕ𝕖𝕞𝕠𝟚 by transforming different synthetic and large real-world ℕ𝔽𝕄s as bit-blasted 𝔽𝕄s. We used existing SAT-based, and BDD approaches to count and uniform random sample configurations. We have shown that ℕ𝕖𝕞𝕠𝟚 can:

- model, extend, automatically optimize, and transform NFMs into the most common formats of 𝔽𝕄s by using the ℕ𝕖𝕞𝕠𝟚 language defined in Listing 4;
- use bit-blasting to encode common types of numerical features and arithmetic constraints;
- represent complex formulas up to 12 bit-width of accuracy without overhead for almost every combination of boolean and arithmetical operations;
- represent real-world ℕ𝔽𝕄s up to colossal sizes without overhead for almost every combination of boolean and arithmetic operations in under 15 min;
- use BDD solver from (Heradio et al., 2022) to uniform random sample configurations up to 10^{45} products in under 10 s; and
- use sharpSAT to count the number of configurations up to 10^{250} products in under 5 h.

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);
- apply expert knowledge to reduce the bit-widths, reducing further the respective ℕ𝔽𝕄 space size. While generally speaking, we would not need the accuracy of thoroughly analyzing the domain of certain ℕ𝔽s (eg., amount of virtual cache (Catenazzi, 2022)), it is not trivial to uncover the exceptions (eg., number of cores (Catenazzi, 2022)). Additionally, it is, again, not trivial how to do it — should we only focus on the domain's lower or upper range? Should we analyze even or odd numbers? We plan to define this in future works.
- run ℕ𝕖𝕞𝕠𝟚 in an ecosystem with different solvers with extended support (eg., attributes, graphical interface); and
- beautify ℕ𝕖𝕞𝕠𝟚's language to be a more human-friendly modeling language.

CRediT authorship contribution statement

Daniel-Jesus Munoz: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing. **Mónica Pinto:** Conceptualization, Investigation, Writing – review & editing, Visualization, Supervision, Funding acquisition. **Lidia Fuentes:** Conceptualization, Methodology, Investigation, Resources, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Don Batory:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

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, Spain funds LEIA UMA18-FEDERJA-15, IRIS PID2021-122812OB-I00 (MCI/AEI), and the PRE2019-087496 grant from the Ministerio de Ciencia e Innovación. Funding for open access charge: Universidad de Málaga / CBUA. Batory is retired, writing free textbooks (Batory, 2021), and is walking dogs for wages.

References

Agh, H., García, F., Piattini, M., 2022. A checklist for the evaluation of software process line approaches. Inf. Softw. Technol. 146 (1).

Apel, S., Batory, D., Kästner, C., Saake, G., 2016. Feature-Oriented Software Product Lines. Springer, NY, USA.

Audemard, G., Simon, L., 2018. On the glucose SAT solver. Int. J. Artif. Intell. Tools 27 (01), 1840001.

Bąk, K., Czarnecki, K., Wąsowski, A., 2010. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In: SLE. pp. 2–9.

Barrett, C., 2013. Decision Procedures: An Algorithmic Point of View, Springer-Verlag, 2008. J. Automat. Reason. 51 (4).

Barrett, C., Tinelli, C., 2018. Satisfiability Modulo Theories. In: Handbook of Model Checking. Springer, NY, USA, pp. 1–2.

Batory, D., 2005. Feature Models, Grammars, and Propositional Formulas. In: SPLC. pp. 2–8.

Batory, D., 2021. Automated Software Design, Vol. 1. Lulu.com.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. Inf. Syst. 35 (6).

Benavides, D., Trinidad, P., Cortés, A., 2007. Automated Reasoning on Feature Models. In: CAISE.

Berger, T., Collet, P., 2019. Usage scenarios for a common feature modeling language. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B. SPLC '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450366687, pp. 174–181. http://dx.doi.org/10.1145/3307630.3342403.

Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K., 2013. A Study of Variability Models and Languages in the Systems Software Domain. IEEE TSE 39 (12).

Biere, A., Heule, M., van Maaren, H., 2009. Handbook of Satisfiability. IOS Press, IEEE.

Brummayer, R., Biere, A., 2009. Boolector: An Efficient SMT Solver for Bit-vectors and Arrays. In: TACAS. Springer, NY, USA, pp. 1–4.

Bryant, R., 2018. Binary Decision Diagrams. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (Eds.), Handbook of Model Checking. Springer, pp. 1–2.

Bryant, R., et al., 2007. Deciding Bit-vector Arithmetic with Abstraction. In: TACAS. Springer, NY, USA, pp. 5–13.

Catenazzi, G., 2022. LKDDb: Linux kernel driver DataBase. https://cateee.net.

Chakraborty, S., et al., 2021. Approximate model counting. FRONTIERS.

Chistikov, D., Dimitrova, R., Majumdar, R., 2017. Approximate Counting in SMT and Value Estimation for Probabilistic Programs. Acta Inform. 54 (8).

Czarnecki, K., Pietroszek, K., 2006. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In: GPCE.

Döller, V., Karagiannis, D., 2021. Formalizing Conceptual Modeling Methods with MetaMorph. In: Enterprise, Business-Process and Information Systems Modeling. Springer, Springer, pp. 4–14.

Foundation, T., 2018. Kconfig Tool Specification. https://www.kernel.org/doc/Documentation/kbuild/kconfig.txt.

Galindo, J.A., Alférez, M., Acher, M., Baudry, B., Benavides, D., 2014. A variability-based testing approach for synthesizing video sequences. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. In: ISSTA 2014, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450326452, pp. 293–303. http://dx.doi.org/10.1145/2610384.2610411.

Ganesh, V., Dill, D., 2006. The Simple Theorem Prover (STP) solver. https://stp.github.io/.

Henard, C., Papadakis, M., Harman, M., Traon, Y.L., 2015. Combining Multi-objective Search and Constraint Solving for Configuring Large Software Product Lines. In: SPLC. IEEE Press, NJ, USA, pp. 3–8.

Heradio, R., Fernandez-Amoros, D., Galindo, J., Benavides, D., Batory, D., 2022. Uniform and Scalable Sampling of Highly Configurable Systems. Empirical Softw. Eng..

Horcas, J.M., 2018. WeaFQAs: A software product line approach for customizing and weaving efficient functional quality attributes. phdthesis. Universidad de málaga.

Horcas, J.M., Galindo, J.A., Benavides, D., 2022a. Flamapy: Python-based AAFM framework. https://github.com/flamapy/core.

Horcas, J.M., Galindo, J.A., Pinto, M., Fuentes, L., Benavides, D., 2022b. FM fact label: A configurable and interactive visualization of feature model characterizations. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. SPLC '22, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450392068, pp. 42–45. http://dx.doi.org/10.1145/3503229.3547025.

Horcas, J., Pinto, M., Fuentes, L., 2018. Variability Models for Generating Efficient Configurations of Functional Quality Attributes. IST J. 95.

Horcas, J., Pinto, M., Fuentes, L., 2020. Extensible and modular abstract syntax for feature modeling based on language constructs. In: SPLC. ACM, pp. 1–7.

Kästner, C., et al., 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: OOPSLA, vol. 46. IEEE/ACM, NJ, USA, pp. 5–18.

Kuiter, E., Krieter, S., Sundermann, C., Thüm, T., Saake, G., 2023. Tseitin or not Tseitin? The impact of CNF transformations on feature-model analyses. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. In: ASE '22, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450394758, http://dx.doi.org/10.1145/3551349.3556938.

Liang, J., Ganesh, V., Czarnecki, K., Raman, V., 2015. SAT-based Analysis of Large Real-world Feature Models Is Easy. In: SPLC. IEEE/ACM, NJ, USA, pp. 2–8.

Mannion, M., 2002. Using first-order logic for product line model validation. In: International Conference on Software Product Lines. Springer, pp. 176–187.

Marchezan, L., Rodrigues, E., Assunção, W.K.G., Bernardino, M., Basso, F.P., Carbonell, J., 2022. Software product line scoping: A systematic literature review. J. Syst. Softw. 186.

Moura, L.D., Bjørner, N., 2008. Z3: An Efficient SMT Solver. In: TACAS. Springer, NY, USA, pp. 1–4.

Munoz, D., Oh, J., Pinto, M., Fuentes, L., Batory, D., 2019a. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In: SPLC. pp. 1–13.

Munoz, D.-J., Oh, J., Pinto, M., Fuentes, L., Batory, D., 2022. A tool to transform feature models with numerical features and arithmetic constraints. In: Perrouin, G., Moha, N., Seriai, A.-D. (Eds.), Reuse and Software Quality. Springer International Publishing, Cham, ISBN: 978-3-031-08129-3, pp. 59–75.

Munoz, D.-J., Pinto, M., Fuentes, L., 2018. Finding correlations of features affecting energy consumption and performance of web servers using the HADAS eco-assistant. Computing 100 (11), 1155–1173.

Munoz, D., Pinto, M., Fuentes, L., 2019. HADAS: Analysing Quality Attributes of Software Configurations. In: SPLC. SPLC '19, ACM, pp. 1–4.

Munoz, D., et al., 2021. Category Theory Framework for Variability Models with Non-functional Requirements. In: CAiSE. Springer International Publishing, pp. 6–12.

Oh, J., Batory, D., Heradio, R., 2024. Finding near-optimal configurations in colossal spaces with statistical guarantees. ACM TOSEM.

Oh, J., Batory, D., Myers, M., Siegmund, N., 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In: ESEC/FSE. pp. 3–9.

Oh, J., Gazillo, P., Batory, D., Heule, M., Myers, M., 2019. Uniform Sampling from Kconfig Feature Models. Tech. Rep. TR-19-02, University of Texas at Austin, Department of Computer Science.

Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I., 2019. Product Sampling for Product Lines: The Scalability Challenge. In: SPLC. SPLC '19, Association for Computing Machinery, pp. 3–5.

Phan, Q., 2015. Model Counting Modulo Theories (Ph.D. thesis). Queen Mary University of London.

Raatikainen, M., Tiihonen, J., Mannisto, T., 2019. Software Product Lines and Variability Modeling: A Tertiary Study. J. Syst. Softw. 149.

Romano, D., Feichtinger, K., Beuche, D., Ryssel, U., Rabiser, R., 2022. Bridging the gap between academia and industry: Transforming the universal variability language to pure::Variants and back. In: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. Association for Computing Machinery, New York, NY, USA, ISBN: 9781450392068, pp. 123–131.

Rossi, F., Beek, P.V., Walsh, T., 2006. Handbook of Constraint Programming. Elsevier.

Schmitt, A., Wiersch, S., Weis, S., 2015. Glencoe-a Visualization Prototyping Framework. In: ICCE. pp. 177–180.

Shi, K., 2017. Combining Evolutionary Algorithms with Constraint Solving for Configuration Optimization. In: ICSME. IEEE/ACM, pp. 3–4.

Shih, F.Y., Cheng, S., 2005. Improved feature reduction in input and feature spaces. Pattern Recognit. (ISSN: 0031-3203) 38 (5), 651–659. http://dx.doi.org/10.1016/j.patcog.2004.10.004.

Siegmund, N., Grebhahn, A., Apel, S., Kästner, C., 2015. Performance-influence Models for Highly Configurable Systems. In: FSE. ACM, New York, NY, USA, pp. 2–10.

Sorensson, N., Een, N., 2005. Minisat V1. 13-a Sat Solver with Conflict-clause Minimization. SAT 2005 (53).

Sundermann, C., Feichtinger, K., 2021. Universal-Variability-Language/uvl-models: SPLC'21 Publication. http://dx.doi.org/10.5281/zenodo.5031829.

Sundermann, C., Feichtinger, K., Engelhardt, D., Rabiser, R., Thüm, T., 2021s. Yet another textual variability language? A community effort towards a unified language. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume a. SPLC '21, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450384698, pp. 136–147. http://dx.doi.org/10.1145/3461001.3471145.

Sundermann, C., Heß, T., Engelhardt, D., Arens, R., Herschel, J., Jedelhauser, K., Jutz, B., Krieter, S., Schaefer, I., 2021b. Integration of UVL in FeatureIDE. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B. SPLC '21, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450384704, pp. 73–79. http://dx.doi.org/10.1145/3461002.3473940.

Sundermann, C., Nieke, M., Bittner, P.M., Heß, T., Thüm, T., Schaefer, I., 2021c. Applications of #SAT Solvers on Feature Models. In: VaMoS. ACM, NY, USA, pp. 3–8.

Systems, C., 2012. Sample Size Calc. https://www.surveysystem.com/sscalc.htm.

Thüm, T., et al., 2014. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. Sci. Comput. Programm. 79.

Thurley, M., 2006. SharpSAT–counting Models with Advanced Component Caching and Implicit BCP. In: SAT. Springer Berlin Heidelberg, pp. 2–5.

Tseitin, G., 1983. 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, pp. 1–2.

Daniel-Jesus Munoz received his M.Sc. Degree in Computer Science Engineering from the University of Málaga (Spain) in 2015, and his M.SC. Degree in Astronomy and Astrophysics in 2017 from the Valencian International University (Spain). He is a Ph.D. . Student since 2016 and a lecturer since 2021 at the Department of Computer Science of the University of Malaga. He is a former guest Researcher at the University of Bristol (UK), University of Texas at Austin (USA) and the KTH Royal Institute of Technology in Stockholm (Sweden). His current areas of expertise are Software Product Lines, Variability, Automated Reasoning, Energy Efficiency, Optimization, Category Theory, and Blockchain.

Mónica Pinto is an Associate Professor in the Languages and Computer Science Department at the University of Málaga (Spain). She received the M.Sc. degree in Computer Science in 1998 from the University of Málaga, and her Ph.D. in 2004 from the same University. Her main research areas are Energy-Aware Software Development, Component-based Software Engineering, Aspect-oriented Software Development, Architecture Description Languages, Model-Driven Development and Context-aware Mobile Middlewares.

Lidia Fuentes received her M.Sc. degree in Computer Science from the University of Málaga (Spain) in 1992 and her Ph.D. in Computer Science in 1998 from the same University. She is a Full Professor at the Department of Computer Science of the University of Málaga since 2011 (previously, Lecturer and Associate Professor from 1993). Currently, she is the head of the CAOSD research group. Her main research areas are Energy-Aware Software Development, Aspect-Oriented Software Development, Model-Driven Development, Software Product Lines, Agent- Oriented Software Engineering, Self-adaptive middleware platforms, Architecture Description Languages and Domain Specific Languages.

Don Batory is a Professor Emeritus in the Department of Computer Science at The University of Texas at Austin. He received a B.S. (1975) and M.Sc. (1977) degrees from Case Institute of Technology, and a Ph.D. (1980) from the University of Toronto. He was a faculty member at the University of Florida in 1981 before he joined the University of Texas in 1983. He was Associate Editor of IEEE Transactions on Software Engineering (1999–2002), Associate Editor of ACM Transactions on Database Systems (1986–1992), member of the ACM Software Systems Award Committee (1989–1993; Committee Chairman in 1992), Program Co-Chair for the 2002 Generative Programming and Component Engineering Conference. He is a proponent of Feature Oriented Software Development (FOSD) and with colleagues (and former students) has recently authored a textbook on the topic. Since 1993, he and his students have written 11 Award Papers for their work in automated program development. He and Lance Tokuda were awarded the Automated Software Engineering 2013 Most Influential Paper Award on their work on program refactorings.