Abstract. Real-world Software Product Lines (SPLs) need Numerical Feature Models (NFM)s whose features not only have boolean values satisfying boolean constraints, but also have numeric attributes satisfying arithmetic constraints. A key operation on NFM}s 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 NFM}s by bit-blasting, the technique that encodes arithmetic as boolean clauses. Nemo translates NFM}s 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 NFM}s, 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 highlyconfigurable 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].

Real-world SPLs need Numerical Feature Models (NFM{s). 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, NFM}s are FM}s with NFs.
SAT solvers find configurations of FM's, because FM's can be translated to PF's, 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 NFM's, 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 NFM's into PF's [26]. Concretely, bit-blasting [11] encodes numerical values into bits and arithmetic constraints into PF's.

In this paper, we present Nemo (Numbers, features, models) which natively supports NFM's 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 N's into PF's; and
- Experimentally comparing the run-time of Nemo with popular SMT and CP solvers on processing bit-blasted PF's on artificial NFM's and 7 real-world NFM's 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](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 [26]. State-of-the-art tools that convert feature models into PF's are FeatureIDE [11] and Glencoe [35]; both translate a graphically-drawn feature model into a PF in a Conjunctive Normal Form (CNF).

However, real-world SPLs use NFM's that contain both binary features and NF's [17]. An NF has a name N, a type (ie., domain), and range (eg., N ∈ [1, 2, ..., 128]).
NFM$s add arithmetic constraints to the set of propositional connectives. And arithmetic constraints can constrain boolean features and vice-versa.

Two examples of NFM$s are: (1) the HADAS eco-assistant \cite{27} where energy parameters are represented as NFM$s in an integer domain, and propositional connectives and inequalities are present in cross-tree constraints (eg., \texttt{AEScrypto} $\Rightarrow$ \texttt{keySize} $>$ 128) and (2) WeaFQAs \cite{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 \cite{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 NFM$s \cite{26}: equality ($=$), inequalities ($\neq$, $>$, $\geq$), 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 \textit{sign-magnitude} encoding) or signed (ie., binary \textit{two’s complement} encoding) \footnote{Two’s complement negative integer encoding is the binary complement of the positive encoding plus one bit.}. We use the Big-Endian representation \footnote{Big-Endian: An order of bits in which the ‘Big end’ (most significant value in the sequence) is first in the sequence.} 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 PF$s for arithmetic relations on Big-Endian signed integers with a value range of $[-4,3]$ (ie., $n = 3$ bits) where bit-1 is the integer sign:

\[
\begin{align*}
a, b &= < a_1, a_2, \ldots, a_n >, < b_1, b_2, \ldots, b_n > \\
\text{where } a_i, b_i &\in \{0, 1\}; 1 \leq i \leq n
\end{align*}
\]

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 ($\neq$) is a bit-by-bit disjunction of \texttt{XOR}s ($\oplus$) (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 involve an implication in case of $\geq$ (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 $3$
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 (i.e., C0 = 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 (i.e., 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 (a1 and b1). The sign bit of the resulting modulo operation is always 0 (i.e., modulo always returns a positive number).

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

<table>
<thead>
<tr>
<th>Row</th>
<th>Operation</th>
<th>Bit-Blasted Model</th>
<th>Propositional Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(NFa == NFb)</td>
<td>(a1 == b1) ∧ (a2 == b2) ∧ (a3 == b3)</td>
<td>(a1 ⇔ b1) ∧ (a2 ⇔ b2) ∧ (a3 ⇔ b3)</td>
</tr>
<tr>
<td>2</td>
<td>(NFa ≠ NFb)</td>
<td>(a1 ≠ b1) ∨ (a2 ≠ b2) ∨ (a3 ≠ b3)</td>
<td>(a1 ⊕ b1) ∨ (a2 ⊕ b2) ∨ (a3 ⊕ b3)</td>
</tr>
<tr>
<td>3</td>
<td>(NFa &gt; NFb)</td>
<td>(a1 &lt; b1) ∨ ((a1 == b1) ∧ (a2 &gt; b2)) ∨ ((a1 == b1) ∧ (a2 == b2) ∧ (a3 &gt; b3))</td>
<td>(¬a1 ∨ b1) ∨ ((a1 ⇔ b1) ∧ (a2 ∧ ¬b2)) ∨ ((a1 ⇔ b1) ∧ (a2 ⇔ b2) ∧ (a3 ∧ ¬b3))</td>
</tr>
<tr>
<td>4</td>
<td>(NFa ≥ NFb)</td>
<td>(a1 &lt; b1) ∨ ((a1 == b1) ∧ (a2 ≥ b2)) ∨ ((a1 == b1) ∧ (a2 == b2) ∧ (a3 ≥ b3))</td>
<td>(¬a1 ∨ b1) ∨ ((a1 ⇔ b1) ∧ (b2 ⇒ a1)) ∨ ((a1 ⇔ b1) ∧ (a2 ⇔ b2) ∧ (b3 ⇒ a3))</td>
</tr>
<tr>
<td>5</td>
<td>(NFa ± NFb)</td>
<td>S1 ≡ [C1, (a1 ⊕ b1) ⊕ C1] × (a2 ⊕ b2) ⊕ C1, (a2 ⊕ b2) ⊕ C1, (a3 ⊕ b3) ⊕ C1</td>
<td>[(a0 ∧ b0) ∨ ((a1 ∧ b1) ∨ (a1 ∧ b1) ∨ ±)), (a1 ⊕ b1) ∧ (a2 ∧ b2) ∧ (a1 ∧ b1)] ∨ (a0 ⊕ b0) ∨ (a2 ⊕ b2) ∧ (a1 ∧ b1) ∨ ±), (a2 ⊕ b2) ∨ (a1 ∧ b1) ∨ ±), (a3 ⊕ b3) ∨ ±)</td>
</tr>
<tr>
<td>6</td>
<td>(NFa * NFb)</td>
<td>M ≡ NFa + NFb + … + NFa</td>
<td>Too large to represent Apply addition (5th row)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N = times</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a0 ≡ a1 ⊕ b1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(NFa/NFb)</td>
<td></td>
<td>Too large to represent Apply subtraction (5th row)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D ≡ #times penultimate negative value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d3 ≡ a1 ⊕ b1</td>
<td>D times</td>
</tr>
<tr>
<td>8</td>
<td>(NFa % NFb)</td>
<td></td>
<td>Too large to represent Apply subtraction (5th row)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MOD ≡ penultimate negative value</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mod3 ≡ 0</td>
<td>D (7th row) times</td>
</tr>
</tbody>
</table>

The majority of SAT solvers primarily work with PIFS in CNF [9]. None applies the optimal alternative – Tseitin’s CNF transformation with skolemization [13]. It is the fastest known encoding to transform PIFS into a CNF formula while maintaining model equivalence and model count (i.e., not altering the total number of solutions).
Bit-blasting NFM’s is a tough task to perform manually. The current prototype does it automatically including boolean and arithmetic features and constraints.

4 Nemo

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 NFM’s, 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, conjunctions, 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.
Algorithm 1: Nemo Complete Procedure (blue lines of Fig. 1)

**Input:** NFM defined in a .txt file
1. Parse features names;
2. Calculate features types;
3. Calculate NFM's bit-widths;
4. Optimize and register the declared and calculated constraints;
5. Bit-blast the NFM;
6. Transform the bit-blasted NFM into a PF;
7. Transform the PF into its Tseitin CNF form;
8. 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\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}. 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>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c 1</td>
<td>variable A  (variables first)</td>
</tr>
<tr>
<td>c 2</td>
<td>variable B</td>
</tr>
<tr>
<td>c 3</td>
<td>variable C</td>
</tr>
<tr>
<td>p cnf 3 2</td>
<td>header, CNF format, 3 variables, and 2 clauses</td>
</tr>
<tr>
<td>1 3 0</td>
<td>(A or C)  (clauses last)</td>
</tr>
<tr>
<td>3 -2 0</td>
<td>and (C or not B)</td>
</tr>
</tbody>
</table>

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\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}.

4.2 Numerical Feature Modeling in Nemo

Currently, most SPL feature modeling languages are tool-specific\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}, eg., Clafer\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}. For Nemo, we abstract the notion of NFM's defining just two entities as in\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}: generic variables and functions. Then, following the meta-model of\footnote{DIMACS: \url{http://archive.dimacs.rutgers.edu/pub/challenge/satisfiability}}, we can define a NFM as a formula with different domains, where a variable is a feature...
Table 3. Nemo output for expr: (A = B) requires C; A,B ∈ [-1,1], C Boolean

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c 1</td>
<td>Abit1</td>
</tr>
<tr>
<td>c 2</td>
<td>Abit2</td>
</tr>
<tr>
<td>c 3</td>
<td>Bbit1</td>
</tr>
<tr>
<td>c 4</td>
<td>Bbit2</td>
</tr>
<tr>
<td>c 5</td>
<td>Tseitin1</td>
</tr>
<tr>
<td>c 6</td>
<td>Tseitin2</td>
</tr>
<tr>
<td>c 7</td>
<td>Tseitin3</td>
</tr>
<tr>
<td>c 8</td>
<td>C Boolean</td>
</tr>
<tr>
<td>p cnf 8 14</td>
<td>header, cnf format, 8 variables, and 14 clauses</td>
</tr>
<tr>
<td>-2 0</td>
<td>not Abit2</td>
</tr>
<tr>
<td>-4 0</td>
<td>not Bbit2</td>
</tr>
<tr>
<td>-2 -4 -5 0</td>
<td>and (not Abit2 or not Bbit2 or not Tseitin1)</td>
</tr>
<tr>
<td>2 4 -5 0</td>
<td>and (Abit2 or Bbit2 or not Tseitin1)</td>
</tr>
<tr>
<td>2 -4 5 0</td>
<td>and (Abit2 or not Bbit2 or Tseitin1)</td>
</tr>
<tr>
<td>-2 -4 5 0</td>
<td>and (not Abit2 or Bbit2 or Tseitin1)</td>
</tr>
<tr>
<td>-1 -3 -6 0</td>
<td>and (not Abit1 or not Bbit1 or not Tseitin2)</td>
</tr>
<tr>
<td>1 3 -6 0</td>
<td>and (Abit1 or Bbit1 or not Tseitin2)</td>
</tr>
<tr>
<td>1 -3 6 0</td>
<td>and (Abit1 or not Bbit1 or Tseitin2)</td>
</tr>
<tr>
<td>3 -1 6 0</td>
<td>and (Bbit1 or not Abit1 or Tseitin2)</td>
</tr>
<tr>
<td>-6 7 0</td>
<td>and (not Tseitin2 or Tseitin3)</td>
</tr>
<tr>
<td>-5 7 0</td>
<td>and (not Tseitin1 or Tseitin3)</td>
</tr>
<tr>
<td>5 6 -7 0</td>
<td>and (Tseitin1 or Tseitin2 or not Tseitin3)</td>
</tr>
<tr>
<td>8 7 0</td>
<td>and (C or Tseitin3)</td>
</tr>
</tbody>
</table>

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 numerical operators.

Listing 1.1 illustrates most of the types of supported clauses:
```python
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
c (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: \((G \lor F) \land ((A \cdot B) > C)\) requires \((F \lor (E = D))\)

Sequentially, the above means:

1. **A_constant**: a constant natural \(\mathbb{N}\) with a value of 3;
2. **B_natural**: a natural \(\mathbb{N}\) between 0 and 3;
3. **C_natural_2**: a natural \(\mathbb{N}\) between 0 and 3,
4. **D_integer**: an integer \(\mathbb{N}\) between \(-2\) and 1 in two’s complement encoding;
5. **E_enumerated_integer**: an enumerated integer \(\mathbb{N}\) 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: \((G \lor F)\); and
9. An arithmetic constraint: \(((A \cdot B) > C)\) requires \((F \lor (E = D))\).

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). If any value of a \(\mathbb{N}\) is negative, it is considered an **integer**.

Second, Nemo dynamically calculates the minimum bit-width of each \(\mathbb{N}\) to generate the shortest \(\mathbb{P}\). The process is based on the possible values of each \(\mathbb{N}\) (eg., range, enumeration) and the domain; **natural** \(\mathbb{N}\)s and constraints produce smaller \(\mathbb{P}\)s. 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** \(\mathbb{N}\).

Third, Nemo readjusts the previous computed widths based on NFM constraints. Leaving aside boolean features, every \(\mathbb{N}\) involved in operations with other \(\mathbb{N}\)s 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 \( NF \)s 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 \( 2^{bit-width} - 1 \).
- **Multiplication**: The extended bit-width is the original multiplied by the number of multiplication operands plus 1. For instance, \( A \times B \times C = D \) implies that \( bit-width_{updated} = (bit-width_{current} \times 3) + 1 \).

### 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 \((3 \times 2^{bit-width - 1})\) Tseitin variables in the \( NFs \) involved;
6. \( \% \) creates \((14 \times 2^{bit-width - 1})\) Tseitin variables in the \( NFs \) involved; and
7. \( \ast \) creates \((6^{bit-width - 1})\) 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 \( NFM \)s viable for any bit-width?
**RQ2:** Do \( Nemo \) bit-blasted \( NFM \)s 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 SPLs. To count the number of configurations of \( Nemo \) bit-blasted \( NFM \)s, we used sharpSAT \([12]\), the state-of-the-art
model counter for \( \mathbb{F} \)s. 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 \( \mathbb{N} \mathbb{F} \)Ms viable for any bit-width?**

We start analyzing the most complex types of \( \mathbb{N} \mathbb{F} \)M 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 \( \mathbb{N} \mathbb{F} \)s. The first set of 5 \( \mathbb{N} \mathbb{F} \)M constraints that were analyzed are defined by \((A \text{ op } B) = C\) where \( \text{op} \in \{+, -, *, /, \%\} \) 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 \( \mathbb{F} \), and b) the time in seconds to count the configurations of those \( \mathbb{F} \)s 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 \(2\times\) rate. \(\ast\) grows exponentially.
- **Time to count:** \( + \) and \(-\) linearly grow in proportion with the bit-width, keeping below a second for 16 bit-width. \(\ast\), \(/\) and \(\%\) grow exponentially, keeping below 50 seconds for 12 bit-width.

![Fig. 2. Nemo generated clauses and counting time in seconds of arithmetic operations.](image)

As the number of \( \mathbb{N} \mathbb{F} \) variables is proportional to the bit-width, Tseitin’s transformation guarantees a linear increase \( O(3n+1) \). As operation \( \ast \) creates
(2^bit-width) carry-bits, Tseitin’s transformation increase is negligible. Hence, carry-bits are the ones causing the exponential growth.\footnote{Multiplying two bit-vectors could generate a double width one (eg., 2^3 \times 2^3 = 2^6). Those are many carry-bits compared to additions which create a maximum of one.}

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. \((A + B) = C \land (D + E) = F\)
4. \((A + B) = C \land (D + E) = F \land (G + H) = I \land (J + K) = L\)

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's 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.
RQ2: Do Nemo bit-blasted 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 NFM's obtained from [29] and [37]. Table 4 lists them, where each system has a different number of NF's and/or different configuration space size. Henceforth, we use FSE2015 to denote the F's from [37]. We modeled their NFM's independently in Clafer, Z3, and Nemo syntax, and additionally ran Nemo to generate the PF's just for sharpSAT. Considering the bottleneck found in RQ1, if a 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 NFM's (eg., we can represent the even values of $[0, 2^{13}]$ in 12 bits).

<table>
<thead>
<tr>
<th>Type</th>
<th>NFM</th>
<th>Description</th>
<th>#F</th>
<th>#NF's</th>
<th>#Configs</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSE2015</td>
<td>Dune</td>
<td>Multi-grid solver</td>
<td>11</td>
<td>3</td>
<td>2,304</td>
<td>Equation solving</td>
</tr>
<tr>
<td></td>
<td>HSMGP</td>
<td>Stencil-grid solver</td>
<td>14</td>
<td>3</td>
<td>3,456</td>
<td>times</td>
</tr>
<tr>
<td></td>
<td>HiPacc</td>
<td>Image processing</td>
<td>33</td>
<td>2</td>
<td>13,485</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trimesh</td>
<td>Triangle mesh library</td>
<td>13</td>
<td>4</td>
<td>239,360</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Model</th>
<th>Z3</th>
<th>Clafer</th>
<th>sharpSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSE2015</td>
<td>Dune</td>
<td>26.18 s</td>
<td>10.49 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td></td>
<td>HSMGP</td>
<td>40.70 s</td>
<td>13.91 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td></td>
<td>HiPacc</td>
<td>457 s</td>
<td>32.52 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td></td>
<td>Trimesh</td>
<td>Time-out</td>
<td>217.01 s</td>
<td>0.01 s</td>
</tr>
</tbody>
</table>

We compared for the same number of solutions the time to count them in seconds with sharpSAT, and one CP and SMT solvers: Clafer and Z3py, 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/](https://www.clafer.org/)
10 Z3py: [https://github.com/Z3Prover/z3](https://github.com/Z3Prover/z3)
strained model is unsatisfiable. If counting took more than 15 minutes, we considered it a time-out. Table 3 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.

![Fig. 4. Nemo runtime in seconds of arithmetic operations and equations sets.](image)

**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. 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.

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 NFs and basic operations was successfully applied for counting-based optimizations of SPLs [20]. Our work extends the encoding to complete arithmetic, and creates a tool that allows NFMs modeling while automatizing and optimizing all the process.

7 Related Work

Work tackling NFMs is rare [22]. Some considered NFs as classical features with just present/absent states [8,29,15]. Some encoded NFs as alternative features, where each value of a 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 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 [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 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 [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 [13]. 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 NFMs usually reduces reasoning time. However, those beyond the ones implemented in Name 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 FM s, NFM s 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 NFM s, 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).

We developed Nemo, a prototype that automatically optimizes and transforms NFM s to a Tseitin Ff in DIMACS format. Nemo represents NFM s 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 NFM s to Ff s and used existing SAT-based approaches to count configurations. We have shown that Nemo can:

- model, automatically optimize, and transform NFM s by using the Nemo language;
- use bit-blasting to encode common types of numerical features and arithmetic constraints;
- represent NFM s 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 NFM s 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