Tackling Expressive Feature-Modeling Constructs with Pseudo-Boolean d-DNNF Compilation

Chico Sundermann*†, Stefan Vill*, Elias Kuiter‡, Sebastian Krieter†, Thomas Thüm†, and Matthias Tichy*
* Ulm University, Germany † TU Braunschweig, Germany ‡ University of Magdeburg, Germany

***Abstract*—Configurable systems typically consist of reusable assets that have dependencies between each other. To specify such dependencies, feature models are commonly used. As feature models in practice are often complex, automated reasoning is typically employed to analyze the dependencies. Here, the de facto standard is translating the feature model to conjunctive normal form (CNF) to enable employing off-the-shelf tools, such as SAT or #SAT solvers. However, modern feature-modeling dialects often contain constructs, such as cardinality constraints, that are ill-suited for conversion to CNF. This mismatch between the input of reasoning engines and the available feature-modeling dialects limits the applicability of the more expressive constructs. In this work, we shorten this gap between expressive constructs and scalable automated reasoning. Our contribution is twofold: First, we provide a pseudo-Boolean encoding for feature models, which facilitates smaller representations of commonly employed constructs compared to Boolean encoding. Second, we propose a novel method to compile pseudo-Boolean formulas to Boolean d-DNNFs. With the compiled d-DNNFs, we can resort to a plethora of efficient analyses already used in feature modeling. Our empirical evaluation shows that our proposal substantially outperforms the state-of-the-art based on CNF inputs for expressive constructs. For every considered dataset representing different feature models and feature-modeling constructs, the feature models can be significantly faster translated to pseudo-Boolean than to CNF. Overall, deriving d-DNNFs from a feature model with the targeted expressive constraints can be substantially accelerated using our pseudo-Boolean approach. For instance, the Boolean approach only scales for group cardinalities with up-to 13 features while pseudo-Boolean d-DNNF compilation can compile cardinalities with thousands of features. Furthermore, our approach is competitive on feature models with only basic constructs.**

***Index Terms*—product lines, knowledge compilation, d-DNNF, pseudo-Boolean logic, feature models**

I. Introduction

Most aspects of modern life, such as transportation [1]–[3], communication [4], [5], and computers [6], [7], rely on configurable systems. Such systems typically consist of multiple features that can be combined (i.e., configured) according to a set of constraints. Variability languages are used to specify such constraints between features [8], [9]. For instance, a constraint may specify that a feature depends on another feature or that at least one of a group of features needs to be selected. A common specification in variability languages are feature models [9], which define features and constraints (e.g., dependencies) between them [10].

Industrial systems often come with thousands of constraints [2], [6], [11], [12], which mandates the usage of automated analyses [3]. For instance, common analyses are checking whether a given configuration is valid or computing the number of valid configurations [13], [14]. The de facto standard for analyzing feature models is translating them to conjunctive normal form (CNF) then applying standardized reasoning engines, such as SAT [10], [13], [15] or #SAT solvers [2], [16], [17].

There is a mismatch between constraint types in available variability languages and constraints supported by existing reasoning engines. While research on scalable solutions for feature-model analysis is mostly limited to Boolean logic [15], [18], the majority of variability languages support constraints that are hard to represent in CNF [9], [19], [20]. Commonly considered examples for such constructs are group cardinality (i.e., select between n and m features from a group), feature cardinality (i.e., select a single feature between n and m times), or constraints over numeric attributes (e.g., overall power usage of features is limited), or non-Boolean features [9], [21]–[23]. For those constructs, translating to Boolean logic may lead to an exponential increase in formula size [24]. In a survey of ter Beek et al. [9], 11 out 13 variability languages include at least one of such constructs. Consequentially, it is often infeasible to represent and reason about feature models containing those constraints with the standard Boolean approaches [24].

Such expressive constructs are also part of most languages used in practice. In a survey on variability-modeling practices in the industry of Berger et al. [25], pure variants [26] was the most used off-the-shelf tool. Their specification relies on various expressive constructs, including group cardinalities and feature attributes [23]. The tools following in popularity, namely Gears, FeatureIDE [27], [28] and DOPLER [29], also support more expressive constructs.

Several studies also report the mandate of these constructs in real-world projects [20], [24], [30]–[32]. Hubaux et al. [30] conducted a survey with four[1] companies and report that feature cardinalities and attribute constraints are desired. Furthermore, expressive constructs, including group cardinalities, attribute constraints, and feature cardinalities, were also observed in other real-world case studies, such as the embedded system eCos [20], configurable jewelry-rings [31], video conference systems by Cisco [32], loan portfolios [33], and university courses [24].

Scalable solutions for various analyses on expressive constructs are lacking. While solutions for checking satisfiability

[1]The four companies come from the computer hardware, meeting management, and document management application domain [30].

are also available for more expressive formats (e.g., SMT [34], [35] or CP [36], [37]), such techniques have not been yet extended for exact counting or enumeration. Furthermore, the expressive power of these approaches also comes with added computational cost [34], [37], [38]. Both, enumeration and counting, are relevant for a plethora of feature-model analyses [13], [16], [39]. Another issue hindering scalability is that many feature-model analyses rely on more than a single invocation of those complex computations [13]. Knowledge compilation is a strategy for automated reasoning which has recently attracted attention for feature-model analysis [40]–[42]. With knowledge compilation, the feature model is translated to a format (e.g., d-DNNF [43] or BDD [44]) with beneficial properties [45]. These formats can be expensive to compile initially, but after this one-time initial effort, they support more efficient analysis techniques [45]. However, compilation for formats popular in feature-model analysis is currently strictly limited to Boolean formulas [13], [43], [46]–[48], which excludes the more expressive feature-modeling constructs.

In this work, we aim to shorten the gap between expressive feature-modeling constructs and suitable reasoning engines. Our strategy to shorten this gap is twofold. First, we provide pseudo-Boolean encodings for common feature-modeling constructs which are often more natural and substantially smaller than respective Boolean encodings. In particular, we target constructs from the Universal Variability Language (UVL) [21], namely all basic constructs and group cardinalities, feature cardinalities, and attribute constraints. We use UVL as reference as it is widely employed [21], [28], [49]–[53] and covers many constructs common in other variability languages [9], [21], including languages employed in industry [22], [26], [27]. The advantage of pseudo-Boolean formulas is that they allow numeric constants, while still only having to manage Boolean variables. Using this flexibility compared to CNFs facilitates encoding expressive constructs while profiting from the performance advantages of strictly Boolean variables. This allows us to reuse established strategies from Boolean [47], [54] and pseudo-Boolean reasoning [55], [56]. The idea is to find a middle ground between powerful[2] but consequentially more computationally demanding reasoning, such as SMT [34], [35], or CP [36]–[38], and purely Boolean formulas [15]. Second, we introduce pseudo-Boolean d-DNNF compilation, which enables compiling pseudo-Boolean constraints to a Boolean d-DNNF. The resulting d-DNNF can then be used to efficiently perform a plethora of relevant feature-model analyses [13], [40], [41], [57]. For instance, counting and enumeration configurations can be performed in linear time (w.r.t. d-DNNF size) on a d-DNNF [13], [45]. Overall, we provide the following contributions in this work:

- We present *pseudo-Boolean encodings* for several common feature-modeling constructs, including the expressive constructs group cardinality, feature cardinality and attribute constraints (Section III).
- We propose a core algorithm and optimizations for *compiling pseudo-Boolean constraints to d-DNNF* (Section IV).
- We provide *publicly available tooling* for both pseudo-Boolean encoding and pseudo-Boolean d-DNNF compilation.
- We examine the advantages of our approach regarding both encoding and compilation over the state-of-the-art with a *large-scale empirical evaluation* (Section V).

[2]Power with respect to the expressiveness of supported formulas.

II. Background

In this section, we introduce the background required for understanding the main concepts in this work, namely feature models, automated reasoning, d-DNNF compilation, and pseudo-Boolean logic.

A. Feature Models

Feature models are a commonly used formalism to specify the set of valid configurations for a configurable system [58]. Typically, a feature model consists of a feature hierarchy, denoting parent-child relationships with additional cross-tree constraints [14], [58]. Often, non-functional attributes can be attached to features (e.g., a price) [9], [21], [35], [59], [60]. However, the types of parent-child relationships and constraints that are supported vastly differ in literature [9].

Table I shows a taxonomy of different feature-modeling classes. The taxonomy is inspired by the language levels used in the Universal Variability Language (UVL) [21]. We use UVL as reference since UVL is widely used [28], [49]–[52] and covers many common constructs [9], [21]. Furthermore, UVL language levels and our taxonomy aim to categorize the feature-modelling based on complexity for reasoning engines. For instance, basic feature models can be translated to CNF within poly-time and poly-space [61], which is the de facto standard input for SAT [62], [63] or #SAT [54], [64], [65] solvers. In contrast, feature models with different feature types require more powerful reasoning (e.g., SMT [35], [49], [66]). Overall, we differentiate between four different classes, namely basic $\mathbb{B}$, cardinality $\mathbb{C}$, attribute constraints $\mathbb{A}$, and feature types $\mathbb{T}$. The classes differ in the supported group types (i.e., hierarchical parent-child relationships), types of cross-tree constraints (e.g., propositional), and feature types (e.g., Boolean or numeric). In this work, we consider all constructs from $\mathbb{C}$, $\mathbb{A}$, and $\mathbb{T}$ as *expressive* constructs. Furthermore, we use $G_{\mathbb{X}}$ to denote the group types introduced by the class $\mathbb{X}$ and $C_{\mathbb{X}}$ types of cross-tree constraints introduced by the class $\mathbb{X}$. For instance, the cardinality class $\mathbb{C}$ introduces the group types $G_{\mathbb{C}} = \{\text{group cardinality, feature cardinality}\}$.

Basic feature models are commonly considered, especially in the context of automated reasoning [13], [15], [27], [67], [68]. Figure 1 showcases a basic feature model specifying the valid configurations of a robot vacuum product line. Each `robot vacuum` requires some kind of `obstacle detection` and `extra storage` (*mandatory* flag). Furthermore, the robots may have `maps`, a `mop mode`, and a `camera` (*optional* flag). For the obstacle detection, users may choose between one or more from `sensor`, `AI`, and `physical` (*or* relation). The extra storage can be either a

TABLE I
FEATURE-MODELING CONSTRUCT TAXONOMY INSPIRED BY UVL LANGUAGE LEVELS [21]

Class	Introduced Groups ($G_{\mathbb{X}}$)	Introduced Cross-tree Constraints ($C_{\mathbb{X}}$)	Introduced Feature Types
Basic $\mathbb{B}$	{Alternative, or, mandatory, optional}	Propositional ($\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow$)	{Boolean}
Cardinality $\mathbb{C}$	{group cardinality, feature cardinality}	-	
Attribute Constraints $\mathbb{A}$	-	attribute terms	-
Feature Types $\mathbb{T}$	-	feature terms	{numeric, string}

`dust storage` or a `water storage` (*alternative* relation). In addition to the hierarchy, two cross-tree constraints denote that `AI` and `maps` rely on having a `camera` and that `mop mode` requires a `water storage`. In basic feature models, the typical parent-child relationships are *optional*, *mandatory*, *alternative*, and *or* [21], [27], [67]. Furthermore, cross-tree constraints are limited to propositional logic with common operators.

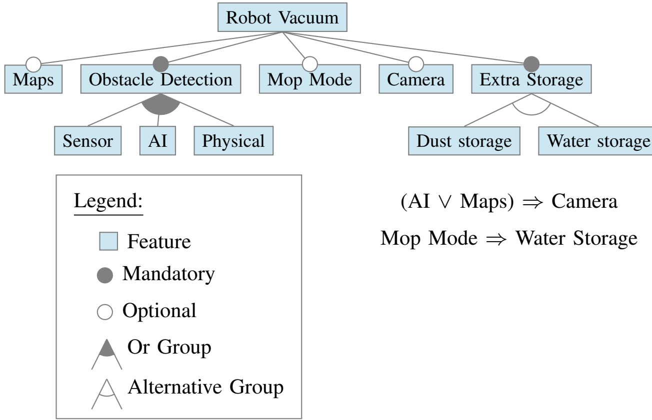

Fig. 1. Basic Feature Model

However, the majority of available variability languages also support at least some more expressive constructs, such as cardinalities, attributes and constraints over them, or typed features [9], [22], [23]. Figure 2 shows an extension of the feature model shown in Figure 1 with constructs from $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{A}$. For obstacle detection, due to a limited number of connections, only one or two features can be selected (*group cardinality*: pick [1..2] out of sensor, AI, and physical). Between one and three extra storage components can now be configured (*feature cardinality*: pick [1..3] incarnations of extra storage). In addition, the features now have non-functional properties, called *feature attributes* [59]. Here, most features have a cost and the storage components also have an attribute indicating the space they require. These attributes can be used for informational purposes or to specify constraints over them. In our example, the overall cost is limited to 15 and the overall space for storages is limited to 6. Various interpretations of feature attribute semantics have been discussed [21], [35], [69]–[71]. We follow UVL [21] regarding the semantics of feature attributes. There is one important difference between feature attributes and typed features: attributes cannot be configured individually (i.e., their value is constant when deriving configurations) [21], [59]. Hence, features are variables while attributes are constants which are typically easier to handle for reasoning engines. Formally, we consider attributes $a_i = (f_i, v_i)$ as always attached to a feature f_i. In a configuration, their value is v_i if its feature f_i is selected and zero otherwise. We discuss the semantics of the presented relationships and constraints in Section III. Nevertheless, we refer to related work on the structure and semantics of feature models for more details [10], [21], [58].

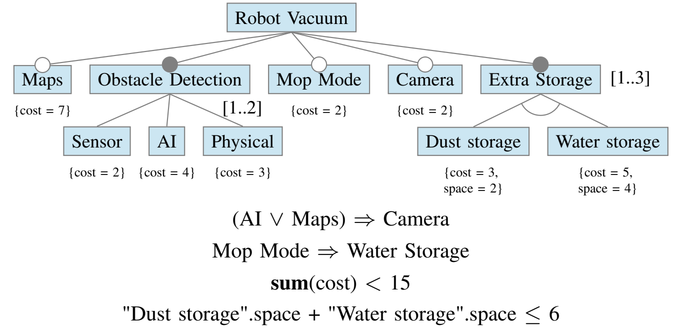

Fig. 2. Feature Model Including Expressive Constructs

B. Logic-Based Reasoning

The de facto standard for feature-model analysis is translating the feature model to Boolean logic [10], [13]–[15], [72], [73] and then invoke off-the-shelf solvers [27], [54], [63], [67], [74]. Here, the features are mapped to Boolean variables and the parent-child relationships and cross-tree constraints translated to Boolean formulas, typically in conjunctive normal form (CNF) [75]. A CNF C is a single conjunction of clauses $C = (C_1 \wedge \ldots \wedge C_n)$. Each clause C_i is a disjunction of literals $C_i = (l_{i,1} \vee \ldots \vee l_{i,m})$. A literal is an atom corresponding to one of the variables v and can be either positive (i.e., v) or negated (i.e., $\bar{v}$). For basic feature models $\mathbb{B}$, translations to CNF are well known and commonly employed [27], [61], [76]. However, constructs from the expressive classes $\mathbb{C}$, $\mathbb{A}$, and $\mathbb{T}$ are not straightforward to translate to CNF. In general, there is a lack of scalable automated reasoning techniques for those expressive constructs.

C. d-DNNF Compilation

Knowledge compilation refers to compiling an original formula (e.g., representing a feature model) to another format with beneficial properties regarding runtime of complex operations [45]. There are various knowledge-compilation target languages that each offer varying tradeoffs between (1) the complexity to compile and (2) operations that are possible in

polynomial time on the compiled artifact [45]. Typical operations are checking satisfiability, model counting, enumerating solutions, or algebraic operations [13], [45].

The *deterministic decomposable negation normal form* (d-DNNF) is a knowledge compilation target language that has gained traction in feature-model analysis over the past years [40], [41], [57], [77]. d-DNNFs support model counting, checking satisfiability, and enumerating solutions within linear time with respect to the d-DNNF's size [45], which can be used to accelerate a plethora of feature-model analyses [13]. None of these operations can be computed in polynomial time on a CNF [45]. The linear runtime operations for d-DNNFs are enabled by the properties *determinism* and *decomposability*. A formula is deterministic if all disjuncts d_i in a disjunction $D = d_1 \vee \ldots \vee d_n$ are pairwise logically contradicting (i.e., $\forall i, j : i \neq j : d_i \wedge d_j \equiv \bot$). A formula is decomposable if no conjunct c_i in a conjunction $C = c_1 \wedge \ldots \wedge c_n$ shares variables with another conjunct (i.e., $\forall i, j : i \neq j : vars(c_i) \cup vars(c_j) = \emptyset$).

Modern d-DNNF compilation is based on the exhaustive Davis-Putnam-Logemann-Loveland (DPLL) algorithm [43], [46], [47]. Here, the solution space of the formula is explored by recursively assigning variables until every solution is found. Algorithm 1 shows the base procedure to compile a given CNF to d-DNNF. There are two main aspects to consider. First, while the formula is not yet satisfied nor unsatisfied, variables are assigned recursively. The subformulas $F|_v$ and $F|_{\neg v}$ resulting from setting v to true and false are recursively compiled to d-DNNF and combined to a disjunction $(F|_v \wedge v) \vee (F|_{\neg v} \wedge \neg v)$. Note that this disjunction is equivalent to F and per construction deterministic as every leftside solution contains v and every rightside solution contains $\neg v$. Second, if the formula can be split into subformulas that share no variables (i.e., components), those components are then processed separately. The compiled sub d-DNNFs can then be conjuncted. Note that the conjunction is equivalent to F and decomposable by construction. Following this procedure, we acquire a semantically equivalent d-DNNF since every conjunction is decomposable and every disjunction is deterministic.

D. Pseudo-Boolean Logic

A linear pseudo-Boolean constraint is an inequality following the structure shown in Equation 1 [55], [78]. Here, k_i and b are constants while x_i are Boolean variables with $x_i \in \{0, 1\}$, where 0 corresponds to false and 1 to true. The set vars(F) denotes the variables considered in the formula. The constants k_i are called *factors* and the rightside b is called *degree* [55]. Each pseudo-Boolean formula is a single conjunction over such constraints. Consequentially, pseudo-Boolean formulas are a generalization of CNFs. A CNF clause $(x_1 \vee \ldots \vee x_n)$ can be represented as $\sum_{i=1}^{n} 1 \cdot x_i \geq 1$. Pseudo-Boolean logic is also closely related to integer linear programming (cf. Section VI for more details). Here, constraints are also linear inequalities, but (1) variables are not limited to Boolean and (2) the goal is typically to optimize an objective function [79].

Algorithm 1 Boolean d-DNNF Compilation

1: **Input:** CNF F
2: **Output:** d-DNNF D
3: **Procedure:** compile(F)
4: **if** satisfied(F) **then**
5:    **return** $\boxed{\top}$
6: **else if** !satisfied(F) **then**
7:    **return** $\boxed{\bot}$
8: **end if**
9: components ← identifyDisconnectedComponents(F)
10: subs ← [ ]
11: **for** C : components **do**
12:    **if** sat(C) **then**
13:       **continue**
14:    **end if**
15:    subs.push(compile($F|_v \wedge v$) ∨ compile($F|_{\neg v} \wedge \neg v$))
16: **end for**
17: **return** $\bigwedge_{S \in \text{subs}} S$

$$\sum_{i=1}^{n} k_i \cdot x_i \geq b \text{ with } k_i, b \in \mathbb{Z}, x_i \in \text{vars}(F) \tag{1}$$

The advantage of pseudo-Boolean is the added expressiveness over CNFs with numerical constants while preserving strictly Boolean variables [55], [78]. With the added expressiveness, certain constructs can be encoded more sparsely (i.e., with fewer literals). In Section III, we showcase these advantages by encoding different feature-modeling constructs in pseudo-Boolean logic. By having strictly Boolean variables, we can reuse strategies (cf. Section IV) that have been optimized over decades of research on Boolean [47], [54], [62] and pseudo-Boolean [55], [56], [80] reasoning.

III. Pseudo-Boolean Feature Model Encoding

To use the benefits of pseudo-Boolean logic for feature modeling, we first need encodings of commonly occuring constructs. While Boolean encodings for feature models are well researched, we only found a Bachelor's thesis by Hennerberg [81] addressing encodings as pseudo-Boolean formulas, but they are limited to basic constructs, group cardinalities, and sums over attributes. Furthermore, their group cardinality encoding produces faulty results if the parent is deselected [81]. Hence, we provide an extended collection of pseudo-Boolean encodings. Note that there are typically multiple valid encodings for a feature-modeling construct. In this work, we limit ourselves to one *semantically equivalent* encoding per construct, for each Boolean logic and pseudo-Boolean logic.

Various feature-modeling constructs have been considered with varying degrees of adoption in practice [9]. With our approach, we limit ourselves to constructs from the Universal Variability Language (UVL) [21] for now. We use UVL as reference since UVL is widely used [28], [49]–[52] and covers many common constructs from relevant languages in research [9], [21] and practice [20], [24], [30]–[32]. Furthermore, the design decisions including supported constructs are

result of a community survey with researchers and practitioners [82], which increases the chances for the constructs to be relevant. With pseudo-Boolean encoding, we aim to find a sweet spot that provides more efficient encodings for several constructs compared to CNF (cf. Section III-A and Section III-B), but can still capitalize on the performance advantages of purely Boolean variables. Hence, we target the first three classes of our taxonomy, namely basic $\mathbb{B}$, cardinality $\mathbb{C}$, and attribute constraints $\mathbb{A}$. The taxonomy classes $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{A}$ cover all UVL language levels but *typed* features [21].

A. Encoding Basic Feature Models

The goal of employing pseudo-Boolean logic for feature models is mainly to provide more efficient encodings (w.r.t. size and translation runtime) for additional expressive constructs. Nevertheless, to translate feature models in practice we initially require encodings for basic constructs. Table II shows Boolean and pseudo-Boolean encodings for the hierarchy group types in features models from the basic class $\mathbb{B}$. Here, $grp^{p}_{c_1..c_n}$ describes the relationship $grp \in \{opt, mand, or, alt\}$ between parent p and its children $\{c_1, \ldots, c_n\}$. Note that a feature c_i can only be child feature in one group, but a parent feature p may have multiple groups. We use the translation implemented in FeatureIDE[3] [27] as reference, which is commonly used [33], [51], [57], [61], [83] and in line with other typical definitions from the literature [74], [76]. For the pseudo-Boolean encoding, our translations are similar to the ones provided by Henneberg [81]. For every pseudo-Boolean encoding, we provide an explanation and give an intuition on its correctness (i.e., its semantical equivalence to the original construct).

TABLE II
ENCODINGS FOR BASIC FEATURE MODEL HIERARCHY

Constraint	Boolean	Pseudo-Boolean
$opt^{p}_{c_1..c_n}$	$\bigwedge_{i=1}^{n}(c_i \Rightarrow p)$	$n \cdot p + \sum_{i=1}^{n} -c_i \geq 0$
$mand^{p}_{c_1..c_n}$	$\bigwedge_{i=1}^{n}(c_i \Leftrightarrow p)$	$n \cdot p + \sum_{i=1}^{n} -c_i = 0$
$or^{p}_{c_1..c_n}$	$p \Leftrightarrow (\bigvee_{i=1}^{n} c_i)$	$n \cdot p + \sum_{i=1}^{n} -c_i \geq 0$ $-p + \sum_{i=1}^{n} c_i \geq 0$
$alt^{p}_{c_1..c_n}$	$p \Leftrightarrow (\bigvee_{i=1}^{n} c_i \wedge \bigwedge_{i<j}(\neg c_i \vee \neg c_j))$	$p + \sum_{i=1}^{n} -c_i = 0$

a) Optional Features: For *optional* features, no constraints are imposed on the child features when the parent is selected. Hence, an optional group only enforces that the selection of a child mandates the selection of the parent feature. Table II shows the Boolean and pseudo-Boolean encoding for an optional group $opt^{p}_{c_1..c_n}$. The pseudo-Boolean constraint is only violated (i.e., left side < 0) if a child is selected but the parent is not. Note that $n \cdot p$ is a valid summand in a pseudo-Boolean constraint as n is a constant and p is a variable.

Correctness. To correctly represent the semantics of optional features, the constraint needs to be only falsified if a child is selected, but the parent is not. For $p = 1$ (i.e., parent selected), the left side is always at least zero, since $\sum_{i=1}^{n} -c_i$ is always at least $-n$. For $p = 0$ (i.e., parent not selected), the left side is always negative if at least one child is selected.

b) Mandatory Features: *Mandatory* features need to be selected if the parent is selected. The pseudo-Boolean constraint for $mand^{p}_{c_1..c_n}$ given in Table II is violated if the parent but not all its children are selected. Note that the provided pseudo-Boolean constraint does not exactly match the definition from Section II-D (i.e., $=$ instead of $\geq$), but can be straightforwardly normalized to the desired form. We discuss normalizations we perform at the end of this section.

Correctness. The mandatory constraint needs to be satisfied in exactly two cases, i.e., the parent is selected with all children, or the parent is deselected and none of its children is selected. Both cases hold since $n \cdot 1 + \sum_{i=1}^{n} -1 = 0$ and $n \cdot 0 + \sum_{i=1}^{n} 0 = 0$. For any case with $p = 1$ and any $c_i = 0$ (i.e., parent without all children), the left side is positive. For any case with $p = 0$ and any $c_i = 1$ (i.e., children without parent), the left side is negative.

c) Or Group: If the parent of an *or* group $or^{p}_{c_1,..c_n}$, is selected, at least one of its child features need to be selected. To specify $or^{p}_{c_1,..c_n}$, we use two pseudo-Boolean constraints that ensure (1) no c_i is selected while p is not and (2) selecting p requires selecting at least one c_i.

Correctness. The semantics of an *or* group are a slight specialization of optional constraints. Since the first constraint is equal to the optional specification, we need to ensure that the second constraint is falsified if the parent but no child is selected. This case holds since $-1 + \sum_{i=1}^{n} 0 < 0$.

d) Alternative Group: If the parent of an *alternative group* $alt^{p}_{c_1..c_n}$ is selected, exactly one of its child features needs to be selected. We encode this behavior by ensuring that the number of selected children is equal to the selection status of the parent (i.e., zero or one).

Correctness. The pseudo-Boolean constraint for $alt^{p}_{c_1..c_n}$ is only satisfied if the parent p and exactly one child c_i (i.e., $1-1=0$) is selected or no parent and no child (i.e., $0-0=0$) is selected.

e) Boolean Cross-Tree Constraints: To translate Boolean cross-tree constraints to pseudo-Boolean logic, we can use well-known translations to CNF and then translate the resulting clauses to pseudo-Boolean constraints. In particular, we convert a clause $(x_1 \vee \ldots \vee x_n)$ to the pseudo-Boolean constraint $\sum_{i=1}^{n} 1 \cdot x_i \geq 1$. The benefit is that we employ available techniques and tools for CNF translation [61].

Correctness. Both the clause $(x_1 \vee \ldots \vee x_n)$ and the pseudo-Boolean constraint $\sum_{i=1}^{n} 1 \cdot x_i \geq 1$ are satisfied if at least one x_i is selected. If no x_i is selected, both the clause and the pseudo-Boolean constraint are falsified. Thus, the clause and the pseudo-Boolean constraint are semantically equivalent.

By combining these translations, we can represent basic feature models in pseudo-Boolean logic. For basic feature models, most constructs can be represented as formulas with similar sizes in Boolean and pseudo-Boolean. The exception are alternatives $alt^{p}_{c_1..c_n}$ which induce considerable difference as we require $\mathcal{O}(n^2)$ clauses in a CNF, but only a single pseudo-Boolean constraint.

[3] https://github.com/FeatureIDE/FeatureIDE. FeatureIDE is a widely used framework for software-product line development [27].

B. Encoding Expressive Constraints

Table III shows Boolean and pseudo-Boolean encodings for constructs in the taxonomy classes cardinality $\mathbb{C}$ and attribute constraints $\mathbb{A}$. As reference for the Boolean encoding, we use UVL conversion strategies [21], [84] which translate the expressive constructs to propositional constraints. We employ the UVL conversion strategies as (1) they are part of the standard UVL tooling and (2) each conversion produces semantically equivalent formulas [21], [84].

TABLE III
ENCODINGS FOR EXPRESSIVE FEATURE-MODELING CONSTRUCTS

Construct	Boolean	Pseudo-Boolean
$[a..b]^p_{c_1..c_n}$	enum($[a..b]^p_{c_1..c_n}$)	$n \cdot p + \sum_{i=1}^{n} -c_i \geq 0$ $-a \cdot p + \sum_{i=1}^{n} c_i \geq 0$ $\sum_{i=1}^{n} -c_i \geq -b$
$f[a..b]$	enum($[a..b]^{cr}_{c_1,..,c_b}$)	$pb([a..b]^{cr}_{c_1,..,c_b})$
$\phi(A) \star \psi(A')$	enum($\phi(A) \star \psi(A')$)	Table IV

a) Group Cardinality: A group cardinality $[a..b]^p_{c_1..c_n}$ specifies that if parent p is selected, between a and b of its children c_i need to be selected. For Boolean, we use the UVL conversion strategy which enumerates the valid assignments over the children $c_1..c_n$ that satisfy the group cardinality. For instance, $[2..3]^p_{\{c_1,c_2,c_3\}}$ would be encoded as $(c_1 \wedge c_2 \wedge c_3) \vee (c_1 \wedge c_2 \wedge \bar{c_3}) \vee (c_1 \wedge \bar{c_2} \wedge c_3) \vee (\bar{c_1} \wedge c_2 \wedge c_3)$. Note that several alternative encodings for at-least-k and at-most-k constraints have been suggested in the literature [24], [85]–[87]. We discuss those encodings and their suitability for d-DNNF compilation in Section VI. The enumeration strategy is also employed by UVL for the remaining expressive constructs [21]. With our pseudo-Boolean encoding shown in the first rows of Table III, we use three constraints that (1) ensure that child features $c_1, .., c_n$ can only be selected if parent p is selected, (2) if p is selected, at least a child features are selected, and (3) at most b child features are selected. Henneberg [81] also provides an encoding for group cardinalities, but his proposal may produce incorrect results if the parent is deselected.

Correctness. The first constraint is equivalent to the optional constraint and ensures that deselecting the parent always requires to deselect all children. So, we can focus on the case the parent is selected (i.e., $p = 1$). The other constraints need to be satisfied if the number of selected children $k = \sum_{i=1}^{n} c_i$ is $a \leq k \leq b$ and falsified if not. Considering the first case (i.e., $a \leq k \leq b$), the second constraint is always satisfied $-a \cdot 1 + k \geq 0$, as well as the third $-k \geq -b$. For $k < a$, $-a \cdot 1 + k \geq 0$ is falsified. For $k > b$, $-k \geq -b$ is falsified.

b) Feature Cardinality: A feature cardinality $f[a..b]$ specifies that a feature f may be selected between a and b times [21], [88]. It is important to consider that for every incarnation $f_{i \in [1,b]}$ of f, the subtree spanned by ancestors of f needs to be cloned as every incarnation f_i can be configured individually [21], [88], [89]. A feature cardinality can therefore be considered as group cardinality $[a..b]^{cr}_{f_1,..,f_b}$ over these clones $f_{i \in [1,b]}$ with a new artificial cardinality root feature cr. To illustrate, Figure 3 shows the converted feature cardinality of `Extra Storage` from our running example given in Figure 2. Instead of a feature cardinality, we have a group cardinality over three clones `Extra Storage`$_{\{1,2,3\}}$. This semantically equivalent interpretation is implicitly used by UVL [21] and explicitly used for our pseudo-Boolean encoding. We use the previously presented strategy to convert the resulting group cardinality ($pb([a..b]^{cr}_{c_1,..,c_b})$ in Table III). We manage cross-tree constraints containing cloned variables in the same way as UVL [21]. The encoding for feature cardinality may construct large formulas with many additional variables created by the clones. The number of variables grows particularly for large upper bounds b, deep hierarchies, and nested feature cardinalities. The latter even results in a multiplicative number of clones. However, this is rather an inherent problem of the individual configurability of the feature incarnations, and not a problem of the encoding itself. It is important to note that other interpretations of feature cardinality have been considered in the literature [90], which do not allow individual configurability of different incarnations of the same feature. However, we stick to the interpretation of feature cardinality used by UVL [21].

Correctness. For the translation from feature cardinality to group cardinality, we rely on established principles [21], [90]. After applying the translation, the correctness is thus reliant on the correctness of the group cardinality encoding, which we have already discussed.

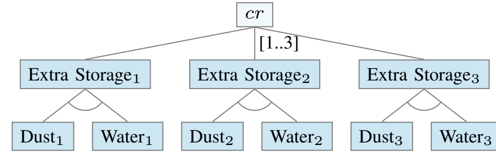

Fig. 3. Example: Conversion of Feature Cardinality to Group Cardinality

c) Numeric Attribute Constraints: In UVL, numeric attribute constraints are terms $\phi(A) \star \psi(A')$ over constants and a set of feature attributes A with $\star \in \{=, \neq, \geq, \leq, >, <\}$. A term $\phi(A)$ is a linear expression over attributes $a_i \in A$ and arithmetic operations (i.e., $+, -, \div, \cdot$). An example is shown in Figure 2 where the space of storage units is limited ("Dust storage".space + "Water storage".space ≤ 6). In that example, we have $\phi(A) =$ ("Dust storage".space + "Water storage".space), $\psi(A') = 6$, and $\star = \leq$. Attributes $a_i = (f_i, v_i)$ resolve to their value v_i if the corresponding feature f_i is selected and to 0 if f_i is deselected. UVL also contains two aggregate functions: sum() and avg(). However, since both can be expanded (e.g., sum(a) to a sum over all instances a_i of that attribute), we do not explicitly consider them here. For the Boolean encoding, we also use the UVL conversion strategies, which enumerate satisfiable assignments over the features f_i corresponding to the respective attributes $a_i \in A \cup A'$ [21]. For the pseudo-Boolean encoding, we propose an approach that *recursively transforms subterms* to pseudo-Boolean logic depending on the connecting operator. We show the different transformations in Table IV.

For unary terms, *constants* c_i and *feature attributes* a_i, the transformation is straightforward. To translate the binary expressions, we always assume that both sides are already

TABLE IV
PSEUDO-BOOLEAN EXPRESSION ENCODINGS

Term	Target Encoding
c_i	c_i
$a_i = (v_i, f_i)$	$v_i \cdot f_i$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) +$ $(c_2 + \sum_{j=1}^{m} l_j \cdot y_j)$	$(c_1 + c_2) + \sum_{i=1}^{n} k_i \cdot x_i$ $+ \sum_{j=1}^{m} l_j \cdot y_j$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) -$ $(c_2 + \sum_{j=1}^{m} l_j \cdot y_j)$	$(c_1 - c_2) + \sum_{i=1}^{n} k_i \cdot x_i$ $+ \sum_{j=1}^{m} -l_j \cdot y_j$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) \cdot$ $(c_2 + \sum_{j=1}^{m} l_j \cdot y_j)$	$(c_1 \cdot c_2) + \sum_{i=1}^{n} c_2 \cdot k_i \cdot x_i$ $+ \sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot z_{i,j}$ $+ \sum_{j=1}^{m} c_1 \cdot l_j \cdot y_j$
$(c_1 + \sum_{i=1}^{n} k_i \cdot x_i) \div$ $(c_2 + \sum_{j=1}^{m} l_j \cdot y_j)$	$\sum_{S \in \mathbb{S}} \frac{c_1}{c_2 + \sum_{s \in S} l_s} \cdot e_S$ $+ \sum_{i=1}^{n} \sum_{S \in \mathbb{S}} \frac{k_i}{c_2 + \sum_{s \in S} l_s} \cdot e_S$

in pseudo-Boolean format (i.e., $(c_1 + \sum_{i=1}^{n} k_i \cdot x_i)$).[4] Otherwise, we can just recursively translate them using the same procedure. For *addition* and *subtraction*, joining the sums and constants already produces a valid pseudo-Boolean expression. When resolving the *multiplication* of both sums, we produce $\sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot x_i \cdot y_j$, which is not in pseudo-Boolean format since x_i and y_j are both variables. Thus, we use a substitution variable $z_{i,j}$ and an additional constraint $z_{i,j} \Leftrightarrow x_i \wedge y_j$ for every pair (x_i, y_j). The added constraint ensures that $z_{i,j}$ behaves equivalently to $x_i \cdot y_i$. For a *division* of two sums, our encoding depends on enumerating the variable selections of the divisor. The main problem lies in the following partial construct after resolving the division: $\frac{k_i \cdot x_i}{c_2 + \sum_{j=1}^{m} l_j \cdot y_j}$. To get the required structure, we create a sum with each element representing the fraction under one variable selection $S \in \mathbb{S}$: $\sum_{S \in \mathbb{S}} \frac{k_i}{c_2 + \sum_{s \in S} l_s} \cdot e_S$. Each substitution variable e_S is equivalent to the corresponding selection S (i.e., $e_S \Leftrightarrow x_i \wedge \bigwedge_{s \in S} y_s \bigwedge_{\bar{s} \in S} \bar{y}_s$). In addition, we create pseudo-Boolean constraints that mark each assignment causing division by zero as unsatisfied (i.e., $S \Leftrightarrow \sum_{j=1}^{m} l_j \cdot y_j \neq -c_2$). Both our encodings for multiplication and division require the introduction of artificial variables, but the encodings preserve quasi-equivalence [61]. Hence, when compiling the resulting formula to d-DNNF, the d-DNNF produces consistent results.

While the unary conversions and the conversions for addition and subtraction roughly keep their original size and structure, the conversions for multiplication and division may produce considerably larger formulas with many artificial variables. Furthermore, each artificial variable requires an additional equivalence constraint. For multiplication, we need to introduce $n \cdot m$ variables and constraints, where n and m are the number of summands in the two multiplied sums. For division, we need to introduce 2^m variables and constraints, where m is the number of summands in the divisor. Consequently, the encodings for multiplication and division may substantially increase the size of the resulting formula, especially for nested multiplications and divisions.

[4]The constraint does not conflict with the structure shown in Equation 1 as the additional constant c_1 can bet set of against the degree.

Correctness. The conversions for unary terms, additions, and subtractions follow straightforward arithmetic principles. Multiplication and division also follow common arithmetic principles, but we need to introduce substitution variables to maintain the form of pseudo-Boolean constraints. For the multiplication conversion to be correct, we need to show that $\sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot z_{i,j}$ behaves equivalently to $\sum_{i=1}^{n} \sum_{j=1}^{m} k_i \cdot l_j \cdot x_i \cdot y_j$ given the added constraint $z_{i,j} \Leftrightarrow (x_i \wedge y_j)$. For $x_i, y_i = 1$ the respective summand resolves to 1 for both variants. For $x_i = 0$ or $y_j = 0$, the summand resolves to 0 for both variants. For division, we need to show that $\sum_{S \in \mathbb{S}} \frac{k_i}{c_2 + \sum_{s \in S} l_s} \cdot e_S$ behaves equivalently to $\frac{k_i \cdot x_i}{c_2 + \sum_{j=1}^{m} l_j \cdot y_j}$ given the added constraints $e_S \Leftrightarrow x_i \wedge \bigwedge_{s \in S} y_s \bigwedge_{\bar{s} \in S} \bar{y}_i$. Any complete assignment resolves to exactly one assignment S' over the divisor variables. Consequently, the respective $e_{S'}$ resolves to 1 (i.e., corresponding to the assignment at hand) and all other e_S resolve to 0. Thus, it remains to show that $\frac{k_i}{c_2 + \sum_{s \in S'} l_s} \cdot e_{S'}$ behaves equivalently to $\frac{k_i \cdot x_i}{c_2 + \sum_{j=1}^{m} l_j \cdot y_j}$. For $x_i = 0$, both variants resolve to 0 due to the added constraint. For $x = 1$, without a loss of generality, assume $S' = \{y_1, y_2, \dots y_p, \bar{y}_{p+1}, \dots, \bar{y}_m\}$. Then, the sums resolve to $\frac{k_i \cdot 1}{c_2 + \sum_{j=1}^{p} l_j \cdot 1 + \sum_{j=p+1}^{m} l_j \cdot 0}$ and $\frac{k_i}{c_2 + \sum_{s \in \{1, \dots, p\}} l_s} \cdot 1$, which are equivalent. Another important aspect regarding quasi-equivalence, is that a substitution variable (e.g., $z_{i,j}$) cannot have two different values for the same assignment over the original variables. This is ensured by the added equivalence constraints that always enforce a value for the substitution variable based on the original variables.

d) Normalizing Pseudo-Boolean Constraints: Some provided encodings slightly deviate from the structure of pseudo-Boolean constraints presented in Section II-D. The definition enforces $\geq$ as comparison operator, but our encodings also use $=$ and attribute constraints in UVL supports all the common comparison operators (i.e., =, $\geq$, $\leq$, $<$, $>$, and $\neq$). For all operators but $\neq$, we use the conversions shown in Table V to conform with the definition. In contrast, we keep $\neq$ constraints and handle them in our compiler (cf. Section IV) as adapting them to match the structure in Equation 1 is complex. Furthermore, we convert floats to integers by multiplying both sides of the inequality with 10^X where x is the highest number of decimal places [55].

TABLE V
CONVERTING COMPARISON OPERATORS

Origin	Normalized
$\sum_{i=1}^{n} a_i \cdot v_i \leq b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b$
$\sum_{i=1}^{n} a_i \cdot v_i > b$	$\sum_{i=1}^{n} a_i \cdot v_i \geq b + 1$
$\sum_{i=1}^{n} a_i \cdot v_i < b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b + 1$
$\sum_{i=1}^{n} a_i \cdot v_i = b$	$\sum_{i=1}^{n} -a_i \cdot v_i \geq -b \wedge \sum_{i=1}^{n} a_i \cdot v_i \geq b$

e) Limitations of Pseudo-Boolean Encoding: With our proposal of employing pseudo-Boolean encoding, we aim to provide reasoning for an extended set of feature-modeling constructs that still allows efficient reasoning. Compared to existing techniques, we substantially push the boundaries of which constructs can be efficiently analyzed, as our empirical evaluation in Section V reveals. Nevertheless, there are several constructs we do not cover that may yield benefits in practice.

Considering the taxonomy of constructs presented in Table I, we provide no encoding for non-Boolean (e.g., numeric) feature types. While numeric feature attributes can be considered as *constants* reliant on the selection status of the feature they are attached to, numeric features behave as *variables*. Thus, they are not supported in pseudo-Boolean encoding [78]. Furthermore, numeric variables cannot be directly used with highly-optimized state-of-the-art d-DNNF compilation techniques, which typically rely on binary branching on Boolean variables as part of DPLL procedures [43], [46], [47], [91]. Including non-Boolean features in reasoning adds a substantial layer of complexity in itself and hinders the reuse of those optimized techniques. As non-Boolean features are considered in various variability languages [9], including UVL [21] and commercial tools [26], reasoning and in particular d-DNNF compilation for such features is a relevant problem setting in practice. While pseudo-Boolean provides no straightforward way to represent numerical variables, it may facilitate techniques to encode numerical variables, such as bit blasting [52]. For instance, considering two 3-bit numbers $a = a_3a_2a_1$ and $b = b_3b_2b_1$, the bit blasting representation of $a > b$ in Boolean logic is $(a_3 \wedge \neg b_3) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \wedge \neg b_2)) \vee ((a_3 \Leftrightarrow b_3) \wedge (a_2 \Leftrightarrow b_2) \wedge (a_1 \wedge \neg b_1))$, which grows quadratically with the number of bits. In pseudo-Boolean logic, it could be represented as $4 \cdot a_3 + 2 \cdot a_2 + 1 \cdot a_1 - 4 \cdot b_3 - 2 \cdot b_2 - 1 \cdot b_1 \geq 1$, which only grows linearly with the number of bits. This idea could be employed in future work to enable numerical variables in pseudo-Boolean logic more efficiently than in Boolean logic.

Few variability languages, such as Clafer [92] also support expressive terms outside the taxonomy we presented (cf. Table I). Examples are Boolean quantifiers (e.g., $\forall$), custom data types similar to objects in programming, or even Turing-complete expressions [9], [92], [93]. Even though it could also be beneficial to support such expressions, reasoning would be considerably more complex.

f) Summary: By combining the different translation strategies, we can now encode all constructs from feature models with basic constructs $\mathbb{B}$, cardinality constructs $\mathbb{C}$, and attribute constraints $\mathbb{A}$ in pseudo-Boolean logic. For various constructs, the proposed pseudo-Boolean encoding is more sparse than the respective Boolean encoding. The differences are especially apparent for cardinalities and addition/subtraction based attribute constraints, where we achieve linear-sized encodings in pseudo-Boolean, but the Boolean encoding grows exponentially in size. Hence, even though our approach does not cover all constructs that could be relevant in practice, it provides more efficient encodings than CNF representations for several widely applied constructs [9], [24], [30], [88]. Note that we examine the correctness and efficiency in practice of our encodings empirically in Section V. Having the encoding enables us to employ pseudo-Boolean based reasoning on UVL models, including our proposal; *pseudo-Boolean d-DNNF compilation* (cf. Section IV).

IV. Pseudo-Boolean d-DNNF Compilation

Our goal is to provide scalable reasoning for common feature-modeling constructs that are hard to analyze with state-of-the-art reasoning engines which typically use CNFs as input [15], [62], [63], [94], [95]. To this end, we propose pseudo-Boolean d-DNNF compilation, which comes with two main advantages. First, while the input is *pseudo-Boolean*, our approach produces *Boolean* d-DNNFs which can be used with available strategies that enable a plethora of feature-model analyses [40], [41], [57]. Second, pseudo-Boolean formulas can provide more sparse encodings for several constructs compared to CNF, while still only having to manage Boolean variables. The limitation to Boolean variables enables us to employ popular techniques from Boolean d-DNNF compilation with only slight adaptations. For instance, exhaustive DPLL [91], which is the algorithmic basis of every popular Boolean d-DNNF compiler [43], [46], [47], is not applicable for non-Boolean variables.

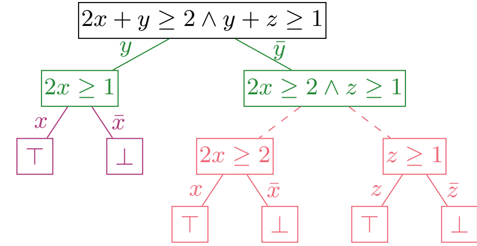

Fig. 4. Trace for Exhaustive Pseudo-Boolean DPLL

A. Reusing Ideas from Boolean Compilation

We propose pseudo-Boolean compilation based on exhaustive DPLL [91] which is also the de facto standard for Boolean compilation (cf. Section II-C) [43], [46], [47]. On the abstraction level of Algorithm 1, the algorithms for pseudo-Boolean and CNF-based d-DNNF compilation behave similarly. Both branch on variables (i.e., set them to true/false) to explore the satisfiability of different assignments. Furthermore, both algorithms use the trace of the branching to construct the d-DNNF. However, many details need to be considered and changed for pseudo-Boolean d-DNNF compilation.

Figure 4 shows an example for our adaptation of exhaustive DPLL to a pseudo-Boolean formula. The adaptation follows previous adaptations of *non-exhaustive* DPLL for pseudo-Boolean formulas [55]. We recursively branch on the variables x, y, z until we have a conflict (i.e., at least one constraint cannot be satisfied by extending the current assignment) or all constraints are satisfied. Each time we assign a variable, we have to update every constraint that contains the assigned variable. For instance, $2x+y \geq 2 \wedge y+z \geq 1$ becomes $2x \geq 1$ under the assignment $y = 1$. When we reach a subformula, where we can split the constraints in a way that they share no variables, we traverse those splits separately as seen in Figure 4 for $2x \geq 2 \wedge z \geq 1$.

As the computed trace (Figure 4) only contains Boolean decisions, we can, analogously to Boolean compilation, derive a d-DNNF from it. Figure 5 shows the resulting d-DNNF. So, we get semantically equivalent results to the state-of-the-art d-DNNF compilation operating on CNF input [47], but benefit from smaller input formulas for several feature-modeling constructs. For every assignment, we create a disjunction,

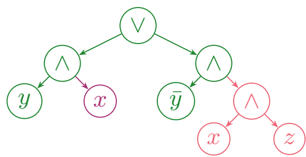

Fig. 5. d-DNNF Corresponding to Traversal in Figure 4

which is, by construction, deterministic. Every time we split the formula due to disconnected components, we create a conjunction which is decomposable by construction. Note that we already propagated the $\top$ and $\bot$ nodes from Figure 4. For instance, we simplified $(x \wedge \top) \vee (\bar{x} \wedge \bot)$ to x.

B. Optimizations

Just as for Boolean d-DNNF compilation, pseudo-Boolean d-DNNF compilation has an exponential runtime complexity in the number of variables. In the worst case, every possible variable assignment needs to be traversed resulting in 2^n branches for n variables. Still, in practice, Boolean d-DNNF compilers scale to various industrial instances [40], [47] by employing various optimizations to efficiently explore the search space [46], [47], [77]. While many optimizations are specifically tailored to the structure of CNFs, adaptations of popular optimizations for CNFs may also be effective for pseudo-Boolean formulas. In the following, we briefly introduce some popular strategies that we have adopted for our pseudo-Boolean approach. Furthermore, we describe the required adaptations.

a) Boolean Constraint Propagation: Boolean constraint propagation refers to automatically assigning variables for which the formula is only satisfiable under that specific assignment. In Boolean logic, this is detected via unit clauses (i.e., clauses with exactly one literal). To satisfy the clause, the corresponding variable has to be assigned such that the literal is satisfied. For pseudo-Boolean logic, we use an adaptation suggested by Chai and Kuehlmann [55] to detect implied literals. Given $\sum_{i=1}^{n} a_i \cdot v_i \geq b$, it can be checked if the largest constant a_{max} is required to satisfy the inequality by checking $\sum_{i=1}^{n} a_i \cdot v_i < b + a_{max}$. In this case, v_{max} is implied.

b) Component Caching: During DPLL traversal, the algorithm often produces the same component (i.e., subformula) multiple times. With component caching, the goal is to save results of components to reuse them later [54], [96]. The main problem here is to efficiently store and detect components, which is typically realized via hashing. Popular hashing schemes used in Boolean solvers are not directly applicable for pseudo-Boolean formulas, because they rely on the structure of CNFs. However, hashing the entire data structure of a subformula has a substantial impact on the performance. We adapt the scheme used by sharpSAT [54] to also include the right side of the pseudo-Boolean inequalities. In particular, we encode a component as $((v_1, ..v_n)(l, r_l)(k, r_k))$ to prepare for hashing where v_i are unassigned variables, l and k are indices of unsatisfied constraints, and r_l and r_k their respective righthand sides. The lists are consistently sorted over the traversal. For instance, we would encode the subformula $2x \geq 2 \wedge y \geq 1$ resulting from setting $\bar{y}$ as $((x), (1, 2), (2, 1))$, where the left side of the second and third tuple refers to the index of the original constraint.

c) Conflict Analysis: In conflict analysis, the partial assignment leading to a conflict is saved as a conflict clause to prevent running into the same conflict multiple times [96]. For instance, after finding that the assignment $\{y, \bar{x}\}$ is unsatisfiable, we could add its negation as conflict clause (i.e., $\neg(y \wedge \bar{x}) \equiv (\bar{y} \vee x)$) to detect this conflict earlier during further branching. Common techniques aim to identify conflict clauses with as few as literals as possible as those can be potentially applied more often [96], [97]. Since assignments are still Boolean in pseudo-Boolean logic and clauses can be easily encoded as pseudo-Boolean constraints, we can use Boolean conflict-driven clause learning (CDCL) [96], [97] with only minor technical adaptations.

d) Variable Orderings: The ordering of variables during branching can have substantial impact on the compilation performance (e.g., regarding runtime) [43], [47]. Various strategies have been proposed that typically rely on the occurrence of variables in regular clauses (e.g., DLCS [98]) and conflict clauses (e.g., VSIDS [94]). For our compiler, we use the same primary variable sorting as `d4` [47]: prioritizing variables that result in disconnected components. In previous work, we observed that `d4` performs best for compiling feature models [13], [99]. Analogous to `d4`, we translate our formula to a hypergraph and use an existing partitioner to derive a cut set, which we then use for ordering. Within the cut set, we order the variables using an adaptation of VSIDS [94] which is commonly used in SAT and #SAT. Our adapted version of VSIDS also considers the impact of a variable in a pseudo-Boolean constraint considering the degree of the formula and the factor of the variable (cf. Section II-D). Furthermore, following insights from preliminary experiments, we use no decay which would decrease the score of variables that are not part of conflict clauses over time [94].

e) In-Compiler Inequality Handling: As hinted in Section III, encoding $\neq$ as pseudo-Boolean constraints following the form $\bigwedge(\sum_{i=1}^{n} k_i \cdot x_i \geq b)$ is complex. One could represent $\sum_{i=1}^{n} k_i \cdot x_i \neq b$ as $\sum_{i=1}^{n} k_i \cdot x_i > b \vee \sum_{i=1}^{n} k_i \cdot x_i < b$, but a disjunction violates the structure of pseudo-Boolean formulas. Hence, instead of translating $\neq$ to a semantically equivalent series of $\geq$ constraints, we allow constraints following the structure $\sum_{i=1}^{n} k_i \cdot x_i \neq b$ in input formulas. The handling of $\neq$ constraints is very similar to $\geq$ with two subtle differences. First, checking the satisfiability during the DPLL traversal needs to be straightforwardly adapted to accomodate the changed semantics between $\neq$ and $\geq$. Second, we cannot apply the Boolean constraint propagation approach used for $\geq$ presented above. Currently, we do not consider inequalities for constraint propagation at all. We assume that decision propagation can only be applied very rarely for $\neq$ constraints as they are generally less restrictive regarding satisfying variable assignments compared to other operators.

C. Tooling

We provide publicly available tooling for both major steps of our pipeline, namely encoding feature models in pseudo-

Boolean logic and pseudo-Boolean d-DNNF compilation. The encoder `UVL2pb`[5] is implemented in Java, so it can be directly used with the UVL parser.[6] The output `.opb` (standard format for pseudo-Boolean formulas) file can then be used as input for our compiler `p2d`[7], whose name is a tribute to the d-DNNF compiler `c2d` that compiles CNFs to d-DNNF. The pseudo-Boolean compiler `p2d` can be used for d-DNNF compilation and for model counting. The compiled d-DNNFs can be saved in the format proposed by `d4` [47], which can then be used as input for further reasoning (e.g., `ddnnife` [40] or `Winston` [57]).

Combining our approach with `ddnnife` enables a pipeline that can perform various feature-model analyses, such as feature-model counting [40], core/dead feature detection [14], [40], and various sampling approaches [40], [41], [100] for UVL models. In particular, the pipeline consists of the following steps which are also depicted in Figure 6. First, `UVL2pb` can be used to translate UVL models to a pseudo-Boolean formula in `.opb`. Since our pseudo-Boolean encoder operates on the metamodel provided by the standard Java-based UVL parser.[8] it can also be integrated in tools using this parser, such as FeatureIDE [27], [28]. Second, the produced `.opb` file can be used as input for `p2d` which compile the d-DNNF. Third, `ddnnife` can be used for reasoning on the compiled d-DNNF either interactively or with one-shot computations.

V. Evaluation

With our empirical evaluation, we examine the benefits of our approach compared to the state-of-the-art. CNF to d-DNNF compilation appears the only option to compile feature models to d-DNNF [40], [41], [57], [77], so we consider it as the main baseline for our evaluation. We use this baseline to evaluate the benefits of pseudo-Boolean representations for encoding and compiling feature models to d-DNNF. To further examine the performance of our approach, we also compare it to a recently published pseudo-Boolean to algebraic decision diagram (ADD) compiler [56]. ADD is another knowledge compilation target language that could be employed for various feature-model analyses (e.g., configuration counting) [56], [101]. We provide a reproduction package including the evaluated tools, subject systems, and final results.[9]

A. Experiment Design

a) Research Questions: To examine the advantages of our approach over the state-of-the-art, we evaluate the performance of the *encoding* ($\mathbf{RQ_1}$) and the performance of the *compilation* ($\mathbf{RQ_2}$, $\mathbf{RQ_3}$). With $\mathbf{RQ_1}$ we examine the performance of our encoder `UVL2pb`, while $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$ target our compiler `p2d`.

$\mathbf{RQ_1}$ How does the performance of pseudo-Boolean encoding compare to CNF encoding for feature models?

$\mathbf{RQ_2}$ How does pseudo-Boolean d-DNNF compilation perform for industrial basic feature models?

- $\mathbf{RQ_{2a}}$ ... compared to Boolean compilation?
- $\mathbf{RQ_{2b}}$ ... compared to compilation into ADDs?

$\mathbf{RQ_3}$ How does pseudo-Boolean d-DNNF compilation perform for expressive constructs?

- $\mathbf{RQ_{3a}}$ ... compared to Boolean compilation?
- $\mathbf{RQ_{3b}}$ ... compared to compilation into ADDs?

For the performance, we consider two dimensions: size of representation w.r.t. to number of literals and runtime for compiling/encoding the input. With $\mathbf{RQ_1}$, we compare the runtimes required to encode the feature models in Boolean and pseudo-Boolean and the sizes of the resulting formulas. In $\mathbf{RQ_2}$, we examine the performance of our approach on feature models with only basic constructs for two reasons. First, while those constructs are used in practice [20], [22]–[24], [29]–[32], there are currently no industrial feature models with expressive constructs *publicly* available. Second, as basic feature-modeling constructs are widely used across systems and application domains [6], [8], [12], [102], [103], we expect that many feature models with more expressive constructs still contain a considerable share of basic constructs. Hence, the performance on basic models is still relevant for feature models containing expressive constructs. Even though there is a lack of publicly available feature models with the expressive constructs, we aim to provide indicative insights on how our approach performs for expressive constructs. With $\mathbf{RQ_3}$, we compare the performance of the pseudo-Boolean and Boolean approach for compiling feature models representing various combinations of constructs.

b) Evaluated Tools: With the evaluation, our main goal is to compare our approach to the Boolean state-of-the-art which is CNF to d-DNNF compilation [13], [47]. Figure 6 provides an overview on the pseudo-Boolean and Boolean pipeline. The contributions of this work are indicated with purple arrows. For our approach, we evaluate our new proposals, namely `UVL2pb` for the pseudo-Boolean encoding and `p2d` for the d-DNNF compilation. For the Boolean baseline, we need a way to translate the models with expressive constructs to CNF. To this end, we use UVL conversion strategies [84] to translate the expressive models to basic models. Then, we use FeatureIDE's CNF translation, as it supports conversion from UVL and is commonly employed in other evaluations [40], [51], [57], [61], [83]. For Boolean d-DNNF compilation, we use `d4`[10] as other evaluations clearly indicate that `d4` performs best for feature-model analyis [40], [99]. For the secondary baseline, we use the recently published pseudo-Boolean to ADD compiler `pbcount` [56] as it appears to be the only available tool for pseudo-Boolean compilation into a knowledge compilation artifact.

c) Subject Systems: Currently, there is a lack of industrial feature models with expressive constructs in the literature. Even though commercial tools support expressive constructs [22], [23], [29] and such constructs are used in practice [30]–[32], feature models created with these tools are not

[5]https://github.com/TUBS-ISF/pseudo-boolean-uvl-encoder
[6]https://github.com/Universal-Variability-Language/uvl-parser
[7]https://github.com/TUBS-ISF/p2d
[8]https://github.com/Universal-Variability-Language/java-fm-metamodel
[9]https://doi.org/10.5281/zenodo.20307775
[10]https://github.com/crillab/d4v2

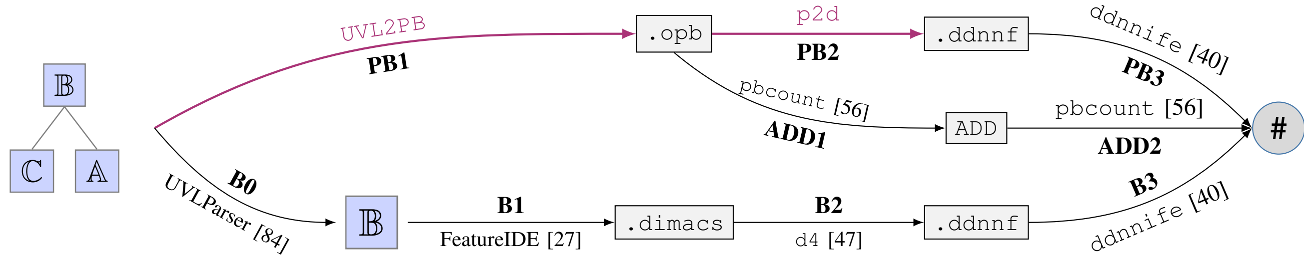

Fig. 6. Comparison of Boolean and Pseudo-Boolean Approach for Compiling Expressive Feature Models to d-DNNF

publicly available. With our selection of subject systems, we aim to provide indicators on the performance of our approach in practice by evaluating on real-world basic feature models and synthesized feature models with expressive constructs. Table VI gives an overview on the different datasets. Each evaluated feature model is given in UVL format.

TABLE VI
OVERVIEW SUBJECT SYSTEMS

Dataset	#Models	#Variables	Constructs	$\mathbf{RQ_i}$
Literature Collection	76	100–80,258	$\mathbb{B}$	$\mathbf{RQ_1}$, $\mathbf{RQ_2}$
Confidential Industry	2	612–645	$\mathbb{B}$ & $\mathbb{A}$	$\mathbf{RQ_1}$, $\mathbf{RQ_3}$
Isolated Construct	2,497	1–8,000	$\mathbb{C}$ & $\mathbb{A}$	$\mathbf{RQ_1}$, $\mathbf{RQ_3}$
Synthesized	2,700	80–2,767	$\mathbb{B}$, $\mathbb{C}$, & $\mathbb{A}$	$\mathbf{RQ_1}$, $\mathbf{RQ_3}$

Real-World Basic Feature Models Most publicly available feature models are limited to basic constructs [8], [11], [12], [17]. In addition, we expect that even feature models with more extended expressions contain a considerable share of basic constraints. Hence, we evaluate our approach on real-world basic feature models. We use a dataset recently collected from industrial models in the literature [12]. The full provided dataset contains several histories of feature models with hundreds of similar feature models. For every history, we use three feature models with the minimum, median, and maximum number of constraints respectively instead of the entire history. A configuration file specifying which models have been used can be found here.[11] The used dataset covers various domains (e.g., automotive and systems software) and varying sizes (100–80,258 variables and 0–388,816 constraints).

Industry Models with Attribute Constraints. Unfortunately, there is a lack of real-world feature models with expressive constructs in the literature. From our industry partner, we have access to two feature models that each contain numerous numeric constraints over feature attributes. Overall, the feature models include constructs from basic feature models $\mathbb{B}$ and attribute constraints $\mathbb{A}$, but no cardinalities ($\mathbb{C}$).

Isolated Construct Models. With the third dataset, we examine the performance of our pseudo-Boolean pipeline for different language constructs in isolation. In particular, we generate feature models including only one of alternative, feature cardinality, group cardinality, and numeric constraints and if needed optional features. Table VII provides an overview on the generated models for the specific constructs. For each incarnation of a specific construct, we create a feature model only including features and constraints corresponding to this incarnation. Depending on the construct, we create different combinations of relevant parameters to achieve a wide coverage. As an example, for alternatives, we create 80 feature models consisting of one alternative with between 100 and 8,000 features each. To evaluate feature cardinality, we create a single feature with that cardinality and attach four groups, each having one type of the *basic* hierarchical groups and three child features. Here, we vary the number of features in the group, the lower bound of the cardinality, and the upper bound. For the numeric constraints, we evaluate constraints consisting of growing expressions A of the respective operators and a constant $\mathbb{E}_p(A)$ for comparison. The constant $\mathbb{E}_p(A)$ controls that a share p of the features need to be selected on average to satisfy the constraint. So for $p = 0.1$, the value $\mathbb{E}_p(A)$ is set to a value that is met or exceeded when selecting 10 % of the features on average.

Synthesized UVL Models. To examine the performance of the different approaches, when dealing with different combinations of the constructs, we randomly generate further UVL models using the UVLGenerator [104].[12] To this end, we aim to cover a wide range of structural properties and distribution of constructs. We generate feature models that include basic constructs and additional expressive constructs by following three strategies: (1) *exactly one* expressive construct, (2) *all but one* expressive construct, (3) all expressive constructs. For each set of included expressive construct, we create 300 feature models which are separated in three size clusters: small, medium, and large. The size clusters are derived from statistics over the collection of real-world feature models [12]. We use the minimal[13] (100), median (554), and third quartile (2,306) number of features from the collection as point of reference for the small, medium, and large models. The generated feature models have the respective reference value +/- 20 % number of features. We configured other structural properties (e.g., tree depth and distribution of group types) in the configurator to match the corresponding distributions of that property in the feature-model collection [12].

d) Experiment Execution: For each of the feature models, we evaluate the whole pseudo-Boolean and Boolean

[11] https://github.com/SoftVarE-Group/feature-model-benchmark/blob/master/pre_configs/minmedmax.json

[12] https://github.com/SoftVarE-Group/uvlgenerator

[13] In the collection [12], only feature models with 100 or more features are considered.

TABLE VII
OVERVIEW ISOLATED EXPRESSIVE CONSTRUCT MODELS

Construct	How?
Group cardinality	$[x..y]^{p}_{c_1,..,c_n}$ with $\forall x, y : x < y$
Feature cardinality	$f[\lceil x \rceil .. \lceil y \rceil]$ with $\forall x, y : x < y$
	$x, y \in \{1, 0.25n, 0.5n, 0.75n, n\}$
	$n \in \{1, 2, \ldots, 100, 200, \ldots 2\,000\}$
Sum	$\sum_{i=1}^{n} a_i \geq b$ with $b = \mathbb{E}_p(\sum_{i=1}^{n} a_i)$
	$n \in \{1, 2, \ldots, 100, 200, \ldots 2\,000\}$
Product	$\prod_{i=1}^{n} a_i + 1 \geq b$ with $b = \mathbb{E}_p(\prod_{i=1}^{n} a_i + 1)$
	$n \in \{2, 3, \ldots, 18\}$
Division	$\frac{a_1 \star \ldots \star a_{n/2}}{a_{n/2+1} \star \ldots \star a_n} > b$ with $b = \mathbb{E}_p(\frac{a_1 \star \ldots \star a_{n/2}}{a_{n/2+1} \star \ldots \star a_n})$
	$n \in \{4, 6, \ldots, 18\}$, $p \in \{0.1, \ldots, 0.9\}$

pipeline (cf. Figure 6). The pseudo-Boolean d-DNNF pipeline consists of translating the UVL model to pseudo-Boolean formula with our encoding (referred to as PB1), compile the formula with `p2d` to d-DNNF (PB2), and compute the model count (i.e., number of satisfying assignments) produced by the d-DNNF with `ddnnife` (PB3). For the Boolean d-DNNF pipeline, we convert the UVL model to a basic feature model using UVL conversion strategies [21] (B0), translate the basic feature model to CNF with FeatureIDE [27] (B1), compile the CNF to d-DNNF with `d4` [47] (B2), and check the model count produced by the d-DNNF with `ddnnife` [40] (B3). The pseudo-Boolean to ADD pipeline consists of translating the UVL model to pseudo-Boolean formula with our encoding (PB1), compile the formula with `pbcount` [56] to an ADD (ADD1), and compute the model count produced by the ADD with `pbcount` [56] (ADD2). The timeouts are ten minutes for conversion to pseudo-Boolean/CNF and ten minutes for compilation. We collect the runtimes for the pipelines, which consist of (PB1 + PB2), (B0 + B1 + B2), and (PB1 + ADD1), respectively. To inspect whether a runtime difference is statistically significant, we use a Wilxocon signed-ranked test [105] and assume the significance level $\alpha = 0.05$. Furthermore, we inspect the sizes of the formula encodings from PB1 and B1 and of the compiled d-DNNFs from PB2 and B2. As indicator for the correctness of our approach, we compare the model counts produced by `ddnnife` for every model (i.e., PB3 vs B3).

In preliminary experiments, we observed that `pbcount` often produces faulty results for our input formulas. After exploring the issue, we found that `pbcount` does not support the whole flexibility of the `opb` format. Instead of producing an error, `pbcount` appears to store a formula based on faulty assumptions. To remedy this, we implemented a preprocessing that sanitizes the input formulas for `pbcount` by removing unsupported syntax.

B. Results

In the following, we present and discuss the results of our empirical evaluation. For every successfully analyzed feature model, the d-DNNFs compiled by `p2d` and `d4` produce the same model count. Since we evaluate on 5,355 models with combinations of constructs, this is a strong indicator that both our pseudo-Boolean encoding and `p2d` are correct.

a) Encoding (RQ1): Figure 7 provides an overview on the performance of the Boolean (CNF) and pseudo-Boolean encoding. The pseudo-Boolean encoding is significantly faster for every dataset, including the basic feature models. The difference in encoding performance is especially apparent for group cardinality, where all isolated models with up-to 5,000 features could be encoded with pseudo-Boolean, but only 7.05 % (up-to 13 features) with the Boolean encoding. Similarly, all sums (up-to 2,100 features) could be encoded with pseudo-Boolean, but only 9.15% (up-to 19 features) for Boolean. For multiplication (94.7% vs 60.9 %) and division (100 % vs 48.1 %), the differences are smaller, but still considerably advantageous for pseudo-Boolean. For the two expressive feature models from our industry partner, the pseudo-Boolean encoding required 1.44 and 1.51 seconds, while Boolean hit the timeout of ten minutes. Comparing the sizes of produced formula which could be encoded by both approaches, alternatives, group cardinalities, and feature cardinalities, are represented with 90 % fewer literals in pseudo-Boolean. In contrast, for multiplication and division the Boolean encoding produces 98.4 % and 83.6 % smaller formulas.

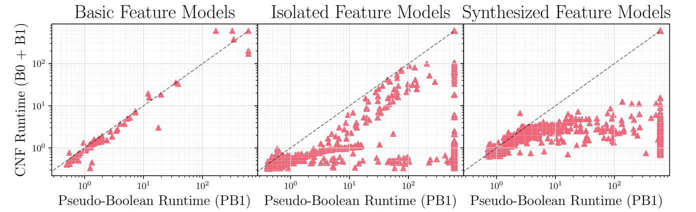

Fig. 7. Boolean vs Pseudo Boolean: Overview Encoding

> **RQ₁** On every evaluated dataset, including basic feature models without any expressive constructs, pseudo-Boolean encoding is significantly faster than Boolean encoding. For some constructs, pseudo-Boolean encoding scales for thousands of features while Boolean encoding hits the timeout of ten minutes already for a few dozen features. The resulting pseudo-Boolean formulas are also smaller for every dataset except multiplications and divisions in isolation.

b) Compiling Basic Feature Models (RQ2): Figure 8 provides a high-level overview on the performance of the three considered compilers `p2d`, `d4`, and `pbcount` with cactus plots over the three main datasets. The runtimes include both compilation and encoding time. For each compiler, the instances are sorted separately according to required runtime. Following conventions in cactus plots, timeouts and runtime errors are omitted. For the literature collection of 76 basic feature models, `d4` scaled to 64 instances, `p2d` to 63, and `pbcount` to 24. For the 2,497 instances representing isolated expressive concepts, `d4` scaled to 678 instances, `p2d` to 2,207, and `pbcount` to 1,886. For the 2,700 random feature models containing different expressive concepts, `d4` scaled to 1,158 instances, `p2d` to 2,626, and `pbcount` to 307.

The results for the literature collection of basic feature models are shown in the first plot in Figure 8. For 8 out of the

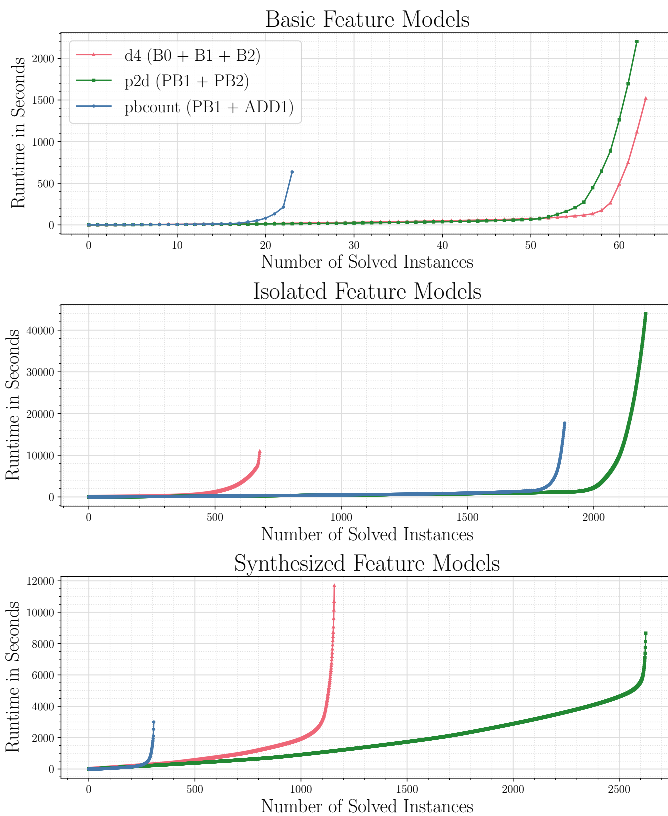

Fig. 8. Overview Performance: Encoding + Compilation

74 successfully encoded feature models, all compilers resulted in a timeout. Respectively, `p2d` failed to compile for three additional models, `d4` for two. `pbcount` only successfully compiled 24 of the 74 encoded models. For the 63 models successfully compiled by the d-DNNF compilers, `d4` requires overall less runtime of 13.6 minutes while `p2d` required 27.2 minutes, but the effect is not significant ($p = 0.09$). Also, `p2d` is faster for 45 models and `d4` for 8. `d4` produces smaller d-DNNFs overall (2,310 vs 2,892 nodes in the median).

RQ$_{2a}$ Our results suggest that our approach is competitive with state-of-the-art Boolean d-DNNF compilation even for basic feature models. Even though `d4` requires less runtime overall for basic feature models, `p2d` is faster for 68.2% of the instances. Nevertheless, it may be beneficial to further optimize our approach for basic feature models regarding compilation time and size of the resulting d-DNNFs.

RQ$_{2b}$ Our compiler `p2d` substantially outperforms `pbcount` on the industrial basic feature models, solving 63 instances within the timeout while `pbcount` only solved 24 instances. While there may still be benefits to applying `pbcount` as ADDs facilitate further efficient queries compared to d-DNNFs, our results suggest that ADD compilation is often not feasible in practice with available compilers.

c) Compiling Expressive Constructs (RQ3): Figure 9 shows the runtimes of the three compilers including encoding times for the *constructs in isolation*. The pseudo-Boolean approach scales for the highest number of features for all expressive constructs and is significantly faster than both `d4` and `pbcount` for each of them ($p < 10^{-5}$). For group cardinality, `p2d` successfully compiled 95.4 % including ones from the largest category (5,000 features) and `pbcount` successfully compiled 82.0 %, while only 7.4 % could be *encoded* in CNF. Nevertheless, each feature model encoded in CNF could be compiled to d-DNNF with `d4`. For additions, the difference is also substantial as 91.7 % (up-to 1,500 features) can be compiled with `p2d` and 84.9% with `pbcount`, but only 9.2 % (up-to 20 features) with the Boolean pipeline. For instances solved by both `p2d` and `d4`, `p2d` overall requires 0.75 minutes while `d4` requires 20.8 minutes. The differences for feature cardinality (`p2d`: 75.0 % vs `d4` 67.3 %), multiplication (84.3 % vs 60.7 %) and division (100 % vs 50.6 %) are smaller but also considerable.

Figure 10 shows the comparison for the different synthesized feature-model datasets that include basic constructs and different combinations of expressive constructs. For example, the dataset *just group cardinality* includes only group cardinality constraints and basic constructs, while the dataset *except cardinality* includes all considered constructs except group cardinality. `p2d` is significantly faster than both `d4` and `pbcount` for every dataset. Over the 2,700 analyzed feature models, the pseudo-Boolean pipeline with `p2d` successfully evaluated 2,626 (97.3 %) and the Boolean pipeline with `d4` 1,158 (42.9 %). Compiling pseudo-Boolean formulas to ADDs with `pbcount` only succeeded for 307 (11.4 %) of the models. For the different datasets, `pbcount` solves at most 18 % of the models (feature cardinality) and as low as 6.0 % (just expressions). Comparing `p2d` and `d4`, the Boolean pipeline with `d4` is reasonably competitive,[14] but still significantly slower for just expressions ($p < 10^{-45}$) and just feature cardinality ($p < 10^{-34}$). For the other datasets, `p2d` scales to substantially more instances: just group cardinality (100 % vs 82.0 %), just aggregate (100 % vs 3.67 %), all but group cardinality (92.7 % vs 2.0 %), all but feature cardinality (94.0 % vs 0.01 %), all but expression (95.3 % vs 8.3%), all constructs (92.7 % vs 6.0 %). For both real-world models from our industry partner, compilation with `p2d` hit the timeout of ten minutes, while for Boolean, already the encoding step failed.

The sizes of compiled d-DNNFs are generally similar, with small advantages for `p2d` on most datasets. However, for multiplication and division the d-DNNFs compiled by `p2d` are considerably larger, probably due to the high number of artificial variables introduced by the encoding. This is especially observable in the isolated models where the d-DNNFs compile in the median to 308 (`p2d`) vs 39 (`d4`) nodes for multiplication and 46 vs 20 nodes for division.

[14] At most three more instances were successfully solved by `p2d`.

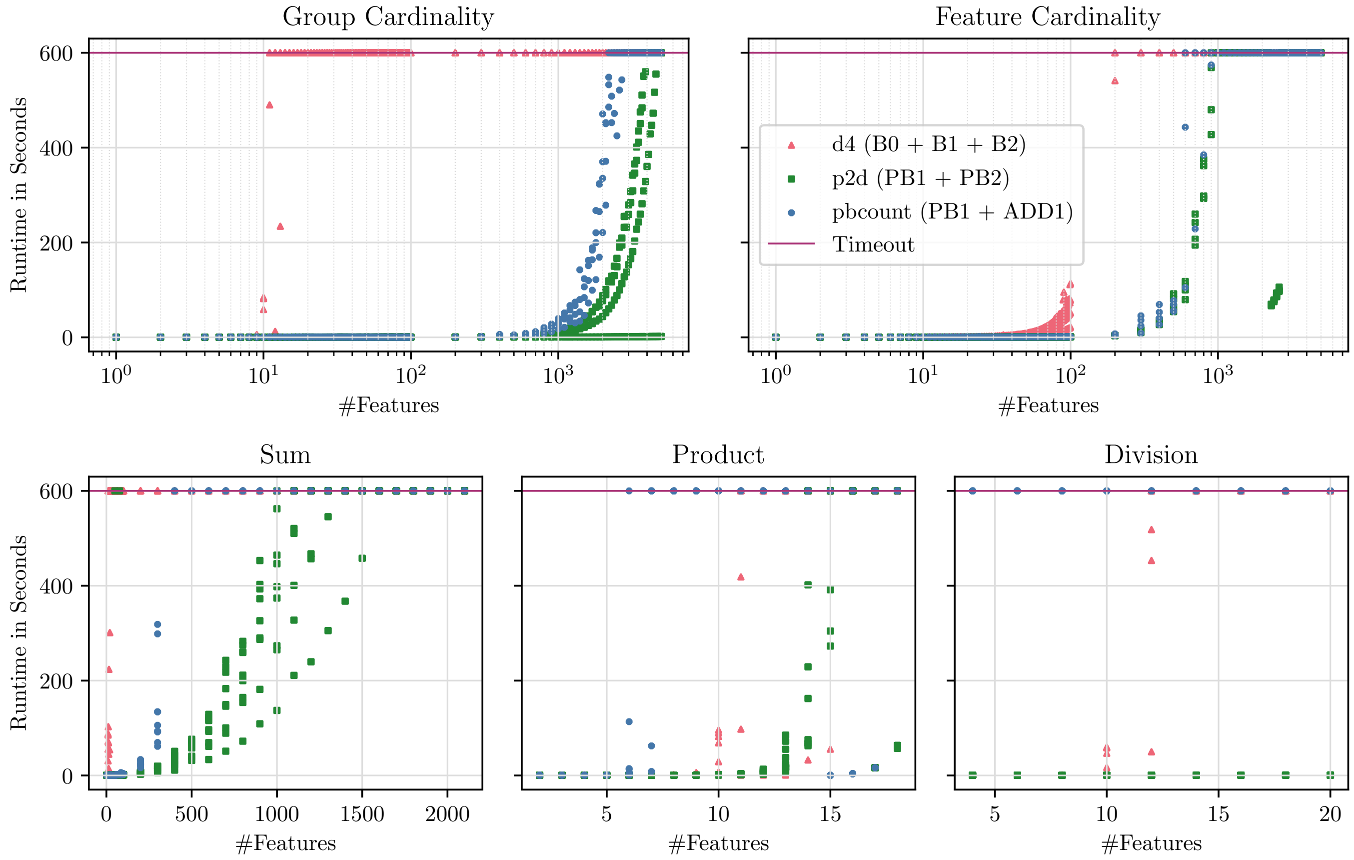

Fig. 9. Isolated Models: Encoding + Compilation

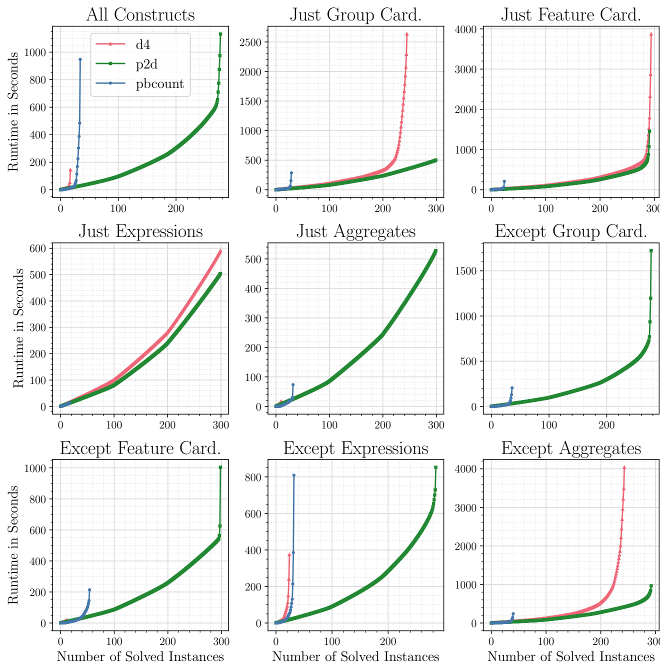

Fig. 10. Synthesized Models: Compilation + Encoding

RQ$_{3a}$ For expressive feature-modeling constructs, pseudo-Boolean d-DNNF compilation yields significant runtime benefits over the Boolean state-of-the-art. Especially if feature models contain aggregate functions or group cardinalities, our pseudo-Boolean approach substantially outperforms the Boolean baseline. Still, the results on the two industrial models and the encoding size for multiplication and division suggest that further optimization on our approach may be beneficial.

RQ$_{3b}$ Our pseudo-Boolean d-DNNF compiler `p2d` substantially outperforms the pseudo-Boolean ADD compiler `pbcount` on the expressive constructs. For the isolated constructs, `pbcount` can take advantage of the rather succinct pseudo-Boolean formulas and, thus, at least outperforms `d4`. Nevertheless, it is still significantly and substantially slower than `p2d`. For the synthesized feature models containing various constructs, `pbcount` fails to scale for most instances (11.4 % successfull) while `p2d` solves the majority (97.3 %).

C. Threats to Validity

a) Computational Bias: The performance of running one of the tools can differ for the same input instances due to computational bias. To reduce such a bias in our conclusions,

we repeated each measurement three times and used the median. The impact is further reduced by the deterministic behavior of each tool, also indicated by the little variance within the three repetitions. Paired with the high number of evaluated models per tool, we do not suppose a considerable impact of computational bias on our conclusions.

b) Conversion to Logic: Alternative strategies to translate the feature models to Boolean and pseudo-Boolean logic may impact the performance [61]. For Boolean logic, a plethora of different encodings for constructs such as at-least-k and at-most-k [24], or numerical variables [52] have been considered. However, their suitability (w.r.t. semantic equivalence) or performance is unknown for encoding feature models and d-DNNF compilation. We discuss this further in Section VI. For pseudo-Boolean logic, we are not aware of alternative strategies in the literature to translate the constructs, but there should be numerous semantically equivalent encodings for the constructs. However, adding more encodings for Boolean or pseudo-Boolean would add another layer of complexity to the already complex evaluation. While out of scope for this work, considering further strategies may be valuable in the future for both Boolean and pseudo-Boolean encoding.

c) Tool Parameters: Both compilers, `d4` [47] and `p2d`, can be parametrized, which can have an impact on the performance. However, considering the combinatorics of different parameters would also vastly increase the complexity of the evaluation and, thus, we consider it out of scope. For the evaluation, we used the default parameters for every considered tool.

d) Transferability of Industrial Models: Our selection of real-world models may not be transferrable to other feature models. However, the collection used is the result of a broad literature survey to identify available real-world models in the literature [12]. The set covers various domains and properties, such as number of features, number of constraints, or tree structure.

e) Transferability of Synthesized Models: The performance results on the synthesized models may not be transferable to practice. However, we are not aware of industrial feature models with the expressive constructs available in the literature. With our synthesized dataset, we aim to showcase the performance for different instances in isolation and also in combination with basic and other expressive constructs. Furthermore, we considered structural properties of industrial feature models to generate the synthesized dataset. Nevertheless, we still consider an evaluation of our approach on more real-world models with expressive constructs as valuable future work.

VI. Related Work

a) Pseudo-Boolean Solving: Some pseudo-Boolean solvers have been proposed [55], [80], [106]. However, these are limited to single-invocation satisfiability checks. Morgado et al. [106] use a pseudo-Boolean SAT solver to count by employing *blocking clauses*. Here, a SAT solver is iteratively queried and returned solutions are negated and added to the formula as blocking clauses. Querying the solver on the updated formula again then always provides a different solution. This procedure is then repeated til all solutions are *blocked* to find the overall number of satisfying assignments. The approach of using blocking clauses however requires SAT calls up-to the number of satisfying assignments on an ever growing formula and is generally seen as not scalable for formulas with many satisfying assignments [107]. Feature models typically induce a very high number of solutions, often with tens, hundreds, or even thousands of digits [12], [99], [108] (e.g., Linux induces more than 10^{616} solutions). Very recently another pseudo-Boolean knowledge compiler has been proposed by Yang and Meel [56]. Their approach compiles to algebraic decision diagrams which are not limited to Boolean variables and are generally more complex to compile. In preliminary experiments, we compared their compiler to `p2d`, but their compiler often produced inconsistent (compared to the results of `p2d` and `d4`) model counts and is likely unsound. Hence, we did not consider their compiler in our empirical evaluation. Nevertheless, the preliminary experiments also showed that `p2d` requires substantially less runtime for compiling the feature models.

b) Integer Programming: Pseudo-Boolean formulas are closely related to popular integer programming variants, such as mixed-integer programming (MIP) or integer linear programming (ILP) [79]. Here, the main difference is that pseudo-Boolean formulas are limited to Boolean variables, while integer programming allows for integer variables. Furthermore, integer programming typically focuses on optimization problems (i.e., minimizing or maximizing a weight function while satisfying a set of constraints) [79]. The limitation of pseudo-Boolean formulas to Boolean variables facilitates the use of many highly optimized techniques, based on Boolean branching which are not directly applicable to integer programming. Thus, we focus on pseudo-Boolean formulas in this work. Nevertheless, the added expressiveness of integer programming may be beneficial for some feature-modeling constructs, such as numeric features, and may be worth exploring in the future. The restriction of integer to Boolean variables has also been considered in integer programming literature, typically referred to as 0-1 integer programming or binary integer programming [79]. Still, to the best of our knowledge, applying d-DNNF compilation to integer programming (including 0-1 integer programming) has not been considered in the literature.

c) d-DNNFs in Feature-Model Analysis: d-DNNFs enable linear time (w.r.t. number of nodes in the d-DNNF) queries for satisfiability, counting, and enumeration [13], [45]. A plethora of feature-model analyses rely on potentially numerous of these operations [13] and, thus, can benefit from d-DNNFs. In related literature d-DNNFs have already been employed for counting [40], [57], sampling [41], [57], satisfiability checks [57], and deriving configurations optimized for different objectives [57].

d) Boolean d-DNNF Compilation: Three Boolean d-DNNF compilers are commonly considered in the literature, namely `c2d` [43], `dSharp` [46], and `d4` [47]. For our empirical evaluation, we only considered `d4` as baseline because the compiler substantially outperformed `c2d` and `dSharp` in previous evaluations on feature models [13], [99]. Our compiler `p2d` adopts many of the strategies employed in the

three Boolean compilers. Still, the type of input formula is vital for the realization of a compiler, as internal data structures and optimizations heavily rely on the exact structure (i.e., CNF for Boolean compilers). Thus, it is not trivial to adapt an existing Boolean d-DNNF compiler to consider pseudo-Boolean formulas.

e) Expressive Feature-Model Dialects: While reasoning for feature-model analysis mostly focuses on Boolean logic in CNF, more expressive constructs (e.g., group cardinality) were considered in a large variety of work. ter Beek et al. [109] provide an overview on textual formats for specifying feature models. 11 out of 13 identified formats support at least one of the expressive constructs facilitated by pseudo-Boolean encoding. Horcas et al. [110] suggest a metamodel approach to cover constructs from different variability languages. Their proposal also includes various constructs, such as arithmetic expressions and cardinalities. In the initial publication of UVL, results of a community questionnaire imply that expressive constructs such as cardinality groups and feature attributes are desired [82]. Various variability-modeling tools that are commonly used in practice [25], such as `pure::variants` [23], DOPLER [29], and Gears [22] also supports more expressive constraints. Furthermore, several case studies on industrial systems imply that the expressive constructs we targeted are mandated [20], [24], [30]–[32]. In summary, insights from related work suggest that more expressive constructs are of interest for practitioners and researchers. The plethora of variability languages, including UVL, can benefit from our advances in scalability for feature-model analysis on these expressive constructs.

f) Encodings for Expressive Feature-Modeling Constructs: While knowledge compilation for expressive feature-modeling constructs has not been yet employed, there has been some works on *encoding* these constructs. Various publications present for cardinality constraints [24], [85], [86] which can be used to model cardinalities. However, many of these encodings only preserve equisatisfiability [24], [85], [86], which can produce faulty results for model counting and for subsequent analyses on a compiled d-DNNF. Furthermore, the performance of these encodings when applied to feature models in general and as input for knowledge compilation is unknown. Nevertheless, examining the plethora of available encodings may reveal suitable and possibly more efficient candidates. We consider the analysis of existing encodings as valuable future work, but as out of scope for this work as it substantially increases the complexity of our evaluation. Benavides et al. [37] propose to encode extended (i.e., attributed) feature models with constraint programming. However, their approach only scales to very small feature models (i.e., fewer than 25 features), while our strategy allows analyzing feature models with thousands of features. Karataş et al. [69] also encode extended feature models with cardinality groups with constraint programming. However, their approach also employs one-shot computations instead of knowledge compilation. Munoz et al. [52] employ bit-blasting to represent numeric features and constraints over them in Boolean logic. However, they did not consider other expressive constructs such as cardinalities. While we did not consider numeric features in this work, the bit-blasting strategy could also be valuable to apply for pseudo-Boolean logic.

g) Reasoning for Expressive Formulas: Satisfiability Modulo Theories (SMT) supports various expressive constructs, such as typed features and constraints over them [34], [66], [111]. SMT solvers are available for satisfiability checking [34], [35] and approximate counting [112]. Constraint programming has been also applied for feature-model analysis [37], [38] to represent basic constructs [37], [38] and attribute constraints [37]. However, for neither of these formats, exact model counters or knowledge compilers are available to the best of our knowledge. Furthermore, the increased expressive power of SMT and CP comes with added computational complexity [37], [38], [113]. With our approach of pseudo-Boolean d-DNNF compilation, we aim to hit a sweet-spot between more complex reasoning on expressive formats and the state-of-the-art, which relies on the rather restrictive format CNF [15], [27], [67], [68].

VII. Conclusion

There is a mismatch between the expressiveness of available variability languages [9] and available reasoning engines [15], [99]. Most variability languages allow various expressions [9] that are not well suited for Boolean encoding. However, state-of-the-art approaches for feature-model analyses reliant on complex computational problems (e.g., configuration counting or t-wise sampling [95]) mainly focus on Boolean logic [13], [15], [27], [52], [67], [68], [103], since scalable solutions for more expressive constructs are often not available. Matching expectations and previous observations [24], our evaluation provides more evidence that encoding common constructs, such as cardinalities or attribute constraints, in Boolean logic comes with substantial scalability issues.

In our work, we tackle this issue by introducing pseudo-Boolean d-DNNF compilation. Our approach consists of a pseudo-Boolean encoding for feature models and a pseudo-Boolean d-DNNF compiler. As the compiled d-DNNF is Boolean, we can employ existing algorithms and tools that enable efficient feature-model analyses [40], [41], [57]. Our empirical evaluation clearly suggests that using our pseudo-Boolean approach yields substantial runtime benefits over the Boolean state-of-the-art. We did not tackle all feature-modeling constructs, as, for instance, we are not able to naturally represent features with other types (e.g., numeric) with pseudo-Boolean logic. Still, pseudo-Boolean d-DNNF compilation considerably advances what feature-modeling constructs can be efficiently analyzed. We envision that the availability of scalable reasoning increases the usage of these constructs, as automated analysis is vital for product-line engineering [2], [3], [14], [102], [114]–[116].

We consider further work on the scalability of expressive feature-modeling constructs as valuable with respect to two dimensions. First, even though our approach of pseudo-Boolean d-DNNF compilation substantially advances the scalability over CNF-based compilation, our empirical evaluation revealed further demand for optimization. Here, it may be promising to develop further heuristics tailored to the pseudo-Boolean structure or develop more efficient encodings for

multiplication and division. Second, further expressive constructs, such as numeric features and constraints over them, are not yet supported by our approach. One approach could be to take advantage of the pseudo-Boolean structure to simplify encodings that are applied for CNFs with a large overhead, such as bit blasting [52]. A more powerful but also more intrusive approach could be to further relax the type of constraints handled by the compiler with designated branching strategies for different types of variables and constraints. Here, variants of the compiler with different constraints enabled could target multiple sweet spots between expressiveness and scalability.

Novelty Statement

Our work reports completely new research results and is not an extension of any existing work. None of our main contributions, namely the pseudo-Boolean encoding, pseudo-Boolean d-DNNF compilation, the tooling, and the empirical evaluation, have been or will be presented or published in other work.

Acknowledgements

This work is based on the thesis of Vill. Many thanks to Jan Baudisch for his support in preparing the reproduction package for the empirical evalution.

References

[1] D. Eichhorn, T. Pett, N. Przigoda, J. Kindsvater, C. Seidl, and I. Schaefer, "Coverage-driven test automation for highly-configurable railway systems," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2023, p. 23–30.

[2] A. Kübler, C. Zengler, and W. Küchlin, "Model Counting in Product Configuration," in *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 2010, pp. 44–53.

[3] O. Oliinyk, K. Petersen, M. Schoelzke, M. Becker, and S. Schneickert, "Structuring Automotive Product lines and Feature Models: An Exploratory Study at Opel," *Requirements Engineering*, vol. 22, no. 1, pp. 105–135, Mar. 2017.

[4] T. F. Bowen, F. S. Dworack, C.-H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin, "The Feature Interaction Problem in Telecommunications Systems," in *Proc. Int'l Conf. on Software Engineering for Telecommunication Switching Systems (SETSS)*. IEEE, 1989, pp. 59–62.

[5] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in Android Testing: They Matter," in *Proc. Int'l Workshop on Advances in Mobile App Analysis (A-Mobile)*. ACM, 2018, pp. 1–6.

[6] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the Linux Kernel Variability Model," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2010, pp. 136–150.

[7] D. Romero-Organvídez, P. Neira, J. A. Galindo, and D. Benavides, "Kconfig Metamodel: A First Approach," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 55–60.

[8] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *IEEE Trans. on Software Engineering (TSE)*, vol. 39, no. 12, pp. 1611–1640, 2013.

[9] M. H. t. Beek, K. Schmid, and H. Eichelberger, "Textual Variability Modeling Languages: An Overview and Considerations," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2019, pp. 151–157.

[10] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2005, pp. 7–20.

[11] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "Is There a Mismatch Between Real-World Feature Models and Product-Line Research?" in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 291–302.

[12] C. Sundermann, V. F. Brancaccio, E. Kuiter, S. Krieter, T. Heß, and T. Thüm, "Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 54–65.

[13] C. Sundermann, E. Kuiter, T. Heß, H. Raab, S. Krieter, and T. Thüm, "On the Benefits of Knowledge Compilation for Feature-Model Analyses," *Annals of Mathematics and Artificial Intelligence (AMAI)*, vol. 92, no. 5, pp. 1013–1050, 2024.

[14] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615–708, 2010.

[15] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "SAT-Based Analysis of Large Real-World Feature Models Is Easy," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 2015, pp. 91–100.

[16] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer, "Applications of #SAT Solvers on Feature Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021.

[17] J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers, "Uniform Sampling From Kconfig Feature Models," University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02, 2019.

[18] C. Sundermann, E. Kuiter, T. Heß, H. Raab, S. Krieter, and T. Thüm, "On the Benefits of Knowledge Compilation for Feature-Model Analyses," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, p. 217.

[19] K. Feichtinger, C. Sundermann, T. Thüm, and R. Rabiser, "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2022, pp. 67–78.

[20] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski, "A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System," in *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2011.

[21] D. Benavides, C. Sundermann, K. Feichtinger, J. A. Galindo, R. Rabiser, and T. Thüm, "UVL: Feature Modelling With the Universal Variability Language," *J. Systems and Software (JSS)*, vol. 225, 2025.

[22] Big Lever Software Inc., "Gears: A Software Product Line Engineering Tool," Website: https://biglever.com/solution/gears/, 2026, accessed: 2026-03-17.

[23] D. Romano, K. Feichtinger, D. Beuche, U. Ryssel, and R. Rabiser, "Bridging the Gap Between Academia and Industry: Transforming the Universal Variability Language to Pure::Variants and Back," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 123–131.

[24] P. M. Bittner, T. Thüm, and I. Schaefer, "SAT Encodings of the At-Most-k Constraint – A Case Study on Configuring University Courses," in *Proc. Int'l Conf. on Software Engineering and Formal Methods (SEFM)*, P. C. Ölveczky and G. Salaün, Eds. Springer, 2019, pp. 127–144.

[25] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A Survey of Variability Modeling in Industrial Practice," in *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2013, pp. 7:1–7:8.

[26] D. Beuche, "Modeling and Building Software Product Lines With pure::variants," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2012, p. 255.

[27] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering Software Variability With FeatureIDE*. Springer, 2017.

[28] C. Sundermann, T. Heß, D. Engelhardt, R. Arens, J. Herschel, K. Jedelhauser, B. Jutz, S. Krieter, and I. Schaefer, "Integration of UVL in FeatureIDE," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2021, pp. 73–79.

[29] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study," *Automated Software Engineering*, vol. 18, no. 1, pp. 77–114, 2011.

[30] A. Hubaux, Q. Boucher, H. Hartmann, R. Michel, and P. Heymans, "Evaluating a textual feature modelling language: Four industrial case

studies," in *Proc. Int'l Conf. on Software Language Engineering (SLE)*. Berlin, Heidelberg: Springer, 2011, pp. 337–356.

[31] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three Cases of Feature-Based Variability Modeling in Industry," in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2014, pp. 302–319.

[32] S. Wang, S. Ali, T. Yue, and M. Liaaen, "Using feature model to support model-based testing of product lines: An industrial case study," in *Proc. Int'l Conf. on Quality Software (QSIC)*, 2013, pp. 75–84.

[33] C. Fritsch, R. Abt, and B. Renz, "The Benefits of a Feature Model in Banking," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2020.

[34] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[35] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer, "SMT-Based Variability Analyses in FeatureIDE," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020.

[36] N. Jussien, G. Rochart, and X. Lorca, "Choco: An Open Source Java Constraint Programming Library," in *Proc. Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP)*. CCSD-HAL, 2008.

[37] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Using Constraint Programming to Reason on Feature Models," in *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2005, pp. 677–682.

[38] R. Pohl, K. Lauenroth, and K. Pohl, "A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 313–322.

[39] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides, "Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2016, pp. 74–78.

[40] C. Sundermann, H. Raab, T. Heß, T. Thüm, and I. Schaefer, "Reusing d-DNNFs for Efficient Feature-Model Counting," *Trans. on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 8, 2024.

[41] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge Compilation Meets Uniform Sampling," in *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 2018, pp. 620–636.

[42] R. Heradio, H. J. Pérez-Morago, D. Fernández-Amorós, R. Bean, F. J. Cabrerizo, C. Cerrada, and E. Herrera-Viedma, "Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models," in *Proc. Int'l Conf. on Intelligent Software Methodologies, Tools and Techniques (SOMET)*. IOS Press, 2016, pp. 139–154.

[43] A. Darwiche, "A Compiler for Deterministic, Decomposable Negation Normal Form," in *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 2002, pp. 627–634.

[44] R. E. Bryant, "Binary Decision Diagrams," in *Handbook of Model Checking*. Springer, 2018, pp. 191–217.

[45] A. Darwiche and P. Marquis, "A Knowledge Compilation Map," *J. Artificial Intelligence Research (JAIR)*, vol. 17, no. 1, pp. 229–264, 2002.

[46] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu, "Dsharp: Fast d-DNNF Compilation with sharpSAT," in *Advances in Artificial Intelligence*, L. Kosseim and D. Inkpen, Eds. Springer, 2012, pp. 356–361.

[47] J.-M. Lagniez and P. Marquis, "An Improved Decision-DNNF Compiler," in *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. International Joint Conferences on Artificial Intelligence, 2017, pp. 667–673.

[48] C. Dubslaff, N. Husung, and N. Käfer, "Configuring BDD Compilation Techniques for Feature Models," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2024, pp. 209–216.

[49] J. Loth, C. Sundermann, T. Schrull, T. Brugger, F. Rieg, and T. Thüm, "UVLS: A Language Server Protocol for UVL," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 43–46.

[50] D. Romero-Organvídez, J. A. Galindo, C. Sundermann, J.-M. Horcas, and D. Benavides, "UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles," *J. Systems and Software (JSS)*, vol. 215, 2024.

[51] K. Feichtinger, J. Stöbich, D. Romano, and R. Rabiser, "TRAVART: An Approach for Transforming Variability Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021, pp. 8:1–8:10.

[52] D.-J. Munoz, M. Pinto, L. Fuentes, and D. Batory, "Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language," *J. Systems and Software (JSS)*, vol. 204, 2023.

[53] H. S. Fadhlillah, K. Feichtinger, P. Bauer, E. Kutsia, and R. Rabiser, "V4rdiac: Tooling for Multidisciplinary Delta-Oriented Variability Management in Cyber-Physical Production Systems," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*, ser. Proc. Int'l Systems and Software Product Line Conf. (SPLC). ACM, 2022, pp. 34–37.

[54] M. Thurley, "sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2006, pp. 424–429.

[55] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 3, pp. 305–317, 2005.

[56] S. Yang and K. S. Meel, "Engineering an Exact Pseudo-Boolean Model Counter," *Proc. Conf. on Artificial Intelligence (AAAI)*, vol. 38, no. 8, pp. 8200–8208, 2024.

[57] P. Bourhis, L. Duchien, J. Dusart, E. Lonca, P. Marquis, and C. Quinton, "Reasoning on Feature Models: Compilation-Based vs. Direct Approaches," Cornell University Library, Tech. Rep., 2023.

[58] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[59] N. Siegmund, S. Sobernig, and S. Apel, "Attributed Variability Models: Outside the Comfort Zone," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 268–278.

[60] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Beyond Boolean Product-Line Model Checking: Dealing With Feature Attributes and Multi-Features," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2013, pp. 472–481.

[61] E. Kuiter, S. Krieter, C. Sundermann, T. Thüm, and G. Saake, "Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2022.

[62] A. Biere, "PicoSAT Essentials," *J. Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[63] D. Le Berre and A. Parrain, "The Sat4j Library, Release 2.2," *J. Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.

[64] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "GANAK: A Scalable Probabilistic Exact Model Counter," in *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, vol. 19. AAAI Press, 2019, pp. 1169–1176.

[65] J. Burchard, T. Schubert, and B. Becker, "Laissez-Faire Caching for Parallel #SAT Solving," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2015, pp. 46–61.

[66] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Comm. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[67] J. A. Galindo, J. M. Horcas, A. Felfernig, D. Fernández-Amorós, and D. Benavides, "FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 16–19.

[68] M. Acher, P. Collet, P. Lahire, and R. B. France, "Familiar: A Domain-Specific Language for Large Scale Management of Feature Models," *Science of Computer Programming (SCP)*, vol. 78, no. 6, pp. 657–681, 2013.

[69] A. S. Karataş, H. Oğuztüzün, and A. Doğru, "From extended feature models to constraint logic programming," *Science of Computer Programming (SCP)*, vol. 78, no. 12, pp. 2295–2312, 2013.

[70] G. Bécan, R. Behjati, A. Gotlieb, and M. Acher, "Synthesis of Attributed Feature Models from Product Descriptions," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. New York, NY, USA: ACM, 2015, pp. 1–10.

[71] L. Ochoa, O. González-Rojas, and T. Thüm, "Using Decision Rules for Solving Conflicts in Extended Feature Models," in *Proc. Int'l Conf. on Software Language Engineering (SLE)*. ACM, 2015, pp. 149–160.

[72] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A Comparison of 10 Sampling Algorithms for Configurable Systems," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.

[73] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?" in *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2019, pp. 240–251.

[74] M. Mendonça, A. Wąsowski, and K. Czarnecki, "SAT-Based Analysis of Feature Models Is Easy," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 2009, pp. 231–240.

[75] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[76] K. Czarnecki and A. Wąsowski, "Feature Diagrams and Logics: There and Back Again," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 2007, pp. 23–34.

[77] C. Sundermann, J. Loth, and T. Thüm, "Efficient Slicing of Feature Models via Projected d-DNNF Compilation," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2024, pp. 1332–1344.

[78] P. L. Hammer and S. Rudeanu, "Pseudo-Boolean Programming," *Operations Research*, vol. 17, no. 2, pp. 233–261, 1969.

[79] L. A. Wolsey, *Integer programming*. John Wiley & Sons, 2020.

[80] H. M. Sheini and K. A. Sakallah, "Pueblo: A hybrid pseudo-boolean sat solver," *J. Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 165–189, 2006.

[81] S. Henneberg, "Next-Generation Feature Models With Pseudo-Boolean SAT Solvers," Bachelor's Thesis, University of Passau, 2011.

[82] C. Sundermann, K. Feichtinger, D. Engelhardt, R. Rabiser, and T. Thüm, "Yet Another Textual Variability Language? A Community Effort Towards a Unified Language," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2021, pp. 136–147.

[83] M. Kowal, S. Ananieva, and T. Thüm, "Explaining Anomalies in Feature Models," in *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2016, pp. 132–143.

[84] C. Sundermann, S. Vill, T. Thüm, K. Feichtinger, P. Agarwal, R. Rabiser, J. A. Galindo, and D. Benavides, "UVLParser: Extending UVL With Language Levels and Conversion Strategies," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2023, pp. 39–42.

[85] C. Sinz, "Towards an optimal cnf encoding of boolean cardinality constraints," in *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2005, pp. 827–831.

[86] O. Bailleux and Y. Boufkhad, "Efficient cnf encoding of boolean cardinality constraints," in *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Berlin, Heidelberg: Springer, 2003, pp. 108–122.

[87] W. Klieber and G. Kwon, "Efficient CNF Encoding for Selecting 1 From n Objects," in *Proc. Int'l Workshop on Constraints in Formal Verification (CFV)*. Springer, 2007.

[88] K. Czarnecki and C. H. P. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," in *Proc. Int'l Workshop on Software Factories (SF)*, 2005, pp. 16–20.

[89] L. Güthing, M. Weiß, I. Schaefer, and M. Lochau, "Sampling Cardinality-Based Feature Models," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. New York, NY, USA: ACM, 2024, pp. 46–55.

[90] C. Quinton, D. Romero, and L. Duchien, "Cardinality-Based Feature Models with Constraints: A Pragmatic Approach," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2013, pp. 162–166.

[91] E. Birnbaum and E. L. Lozinskii, "The Good Old Davis-Putnam Procedure Helps Counting Models," *J. Artificial Intelligence Research (JAIR)*, vol. 10, pp. 457–477, 1999.

[92] P. Juodisius, A. Sarkar, R. Rao Mukkamala, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: Lightweight Modeling of Structure, Behaviour, and Variability," *Computing Research Repository (CoRR)*, 2018.

[93] S. Heisinger, M. Heisinger, and M. Seidl, "(Semantic) Feature Model Differences with (Q)SAT," in *Proc. Int'l Conf. on Software Language Engineering (SLE)*. ACM, 2025, pp. 189–198.

[94] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. Anual Conf. on Design Automation (DAC)*. ACM, 2001, pp. 530–535.

[95] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich, "YASA: Yet Another Sampling Algorithm," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020.

[96] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, "Combining Component Caching and Clause Learning for Effective Model Counting," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2004, pp. 20–28.

[97] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 131–153.

[98] T. Sang, P. Beame, and H. Kautz, "Heuristics for Fast Exact Model Counting," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 226–240.

[99] C. Sundermann, T. Heß, M. Nieke, P. M. Bittner, J. M. Young, T. Thüm, and I. Schaefer, "Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces," *Empirical Software Engineering (EMSE)*, vol. 28, no. 2, p. 38, 2023.

[100] L. Licha, "Cutting Edge T-Wise Sampling With ddnnife," Bachelor's Thesis, University of Ulm, 2023.

[101] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and Their Applications," in *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. IEEE, 1993, pp. 188–191.

[102] M. Hentze, C. Sundermann, T. Thüm, and I. Schaefer, "Quantifying the Variability Mismatch Between Problem and Solution Space," in *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2022, pp. 322–333.

[103] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools," in *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2009, pp. 761–762.

[104] C. Sundermann, T. Heß, R. Sundermann, E. Kuiter, S. Krieter, and T. Thüm, "Generating Feature Models with UVL's Full Expressiveness," in *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 2024, pp. 61–65.

[105] R. F. Woolson, *Wilcoxon Signed-Rank Test*. John Wiley & Sons, 2008.

[106] A. Morgado, P. Matos, V. Manquinho, and J. Marques-Silva, "Counting Models in Integer Domains," in *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2006, pp. 410–423.

[107] T. Toda and T. Soh, "Implementing Efficient All Solutions SAT Solvers," *ACM J. of Experimental Algorithmics (JEA)*, vol. 21, no. 1, pp. 1.12:1–1.12:44, 2016.

[108] E. Kuiter, C. Sundermann, T. Thüm, T. Heß, S. Krieter, and G. Saake, "How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History," *Trans. on Software Engineering and Methodology (TOSEM)*, vol. 35, no. 1, 2025.

[109] M. H. t. Beek, K. Schmid, and H. Eichelberger, "Textual Variability Modeling Languages: An Overview and Considerations," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2019, pp. 151–157.

[110] J.-M. Horcas, M. Pinto, and L. Fuentes, "A modular metamodel and refactoring rules to achieve software product line interoperability," vol. 197, Mar. 2023.

[111] E. G. Karpenkov, K. Friedberger, and D. Beyer, "JavaSMT: A Unified Interface for SMT Solvers in Java," in *Proc. IFIP Working Conf. on Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer, 2016, pp. 139–148.

[112] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in smt and value estimation for probabilistic programs," in *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Berlin, Heidelberg: Springer, 2015, pp. 320–334.

[113] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *Proc. Brazilian Symposium on Formal Methods (SBFM)*. Springer, 2009, pp. 23–36.

[114] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 6:1–6:45, 2014.

[115] R. Heradio, D. Fernández-Amorós, J. A. Cerrada, and I. Abad, "A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models," *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)*, vol. 23, no. 08, pp. 1177–1204, 2013.

[116] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A Classification of Product Sampling for Software Product Lines," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2018, pp. 1–13.