# Reusing d-DNNFs for Efficient Feature-Model Counting

CHICO SUNDERMANN, University of Ulm, Germany
HEIKO RAAB, University of Ulm, Germany
TOBIAS HESS, University of Ulm, Germany
THOMAS THÜM, Paderborn University, Germany
INA SCHAEFER, Karlsruhe Institute of Technology, Germany

Feature models are commonly used to specify valid configurations of a product line. In industry, feature models are often complex due to numerous features and constraints. Thus, a multitude of automated analyses have been proposed. Many of those rely on computing the number of valid configurations, which typically depends on solving a #SAT problem, a computationally expensive operation. Even worse, most counting-based analyses require evaluation for multiple features or partial configurations resulting in numerous #SAT computations on the same feature model. Instead of repetitive computations on highly similar formulas, we aim to improve the performance by reusing knowledge between these computations. In this work, we are the first to propose reusing d-DNNFs for performing repetitive counting queries on features and partial configurations. In our experiments, reusing d-DNNFs saved up-to ~99.98% compared to repetitive invocations of #SAT solvers even when including compilation times. Overall, our tool `ddnnife` combined with the d-DNNF compiler `d4` appears to be the most promising option when dealing with many repetitive feature-model counting queries.

CCS Concepts: • **Computing methodologies** → **Knowledge representation and reasoning**; **Representation of Boolean functions**; • **Software and its engineering** → **Software product lines**; • **Hardware** → *Theorem proving and SAT solving*.

Additional Key Words and Phrases: d-DNNF, feature model, product line, knowledge compilation, model counting, #SAT

**ACM Reference Format:**
Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2024), 30 pages. https://doi.org/10.1145/3680465

## 1 INTRODUCTION

A product line describes a family of related products which share a set of features [7, 9, 63, 96, 100]. Typically, not every configuration of these features is valid (i.e., represents an actual product). Feature models are the de facto standard for specifying the set of valid software configurations [7, 9, 83] and are also frequently used in other domains [32, 34, 35, 47]. In general, a feature model consists of the underlying features and a set of constraints describing dependencies between features. Due to high numbers of features and constraints, it is typically infeasible to analyze feature models manually [9]. Hence, automated analyses, such as checking whether a given configuration is valid, are demanded [2, 9–11, 52, 64, 89].

Numerous practice-relevant analyses depend on computing the number of valid configurations for a feature model [17, 30, 36, 38, 53, 92, 93], such as uniform random sampling [75] and feature prioritization [93]. To employ standardized solvers, feature models are often translated to propositional logic for analysis [21, 66]. While SAT solvers are used for checking the satisfiability, #SAT solvers are dedicated to compute the number of

Authors' addresses: Chico Sundermann, University of Ulm, Ulm, Germany; Heiko Raab, University of Ulm, Ulm, Germany, heiko.raab@uni-ulm.de; Tobias Heß, University of Ulm, Ulm, Germany; Thomas Thüm, Paderborn University, Paderborn, Germany; Ina Schaefer, Karlsruhe Institute of Technology, Karlsruhe, Germany.


1049-331X/2024/1-ART1
https://doi.org/10.1145/3680465

satisfying assignments [53, 94]. Even though #SAT is computationally complex [31, 43, 98, 99], in recent work, we observed [91] that state-of-the-art #SAT solvers can compute the number of valid configurations for many real-world feature models within a few seconds.

For the majority of applications based on feature-model counting, a single #SAT invocation is not sufficient [92]. For example, it can be beneficial to prioritize those features in development that appear in many valid configurations. Using an off-the-shelf #SAT solver as black box, it may be necessary to invoke a separate #SAT computation for every single feature [36, 93]. While several incremental solvers (i.e., that reuse information between queries) for regular SAT were considered in the literature [27, 71], incremental #SAT was not considered yet. As feature models with up to tens of thousands of features have been reported [47, 53, 58, 60, 94], repetitive counting queries without reusing information may require days of computation time [94].

Short response times and reduced resource consumption for feature-model counting are mandated in practice. While long runtimes can be remedied to some degree (e.g., by using parallelization), feature-model analyses are often applied in interactive applications [2, 10, 32, 52, 64] or continuous-integration environments [59, 79]. For instance, many analyses based on feature-model counting can help to understand the configuration space after a change [93] which is relevant for industrial applications [12, 13]. Long response times for analyzing the changes may severely hinder the workflow. We also observed the mandate for short response times from collaborations with the automotive industry, where we already integrated our proposal to improve scalability.

With ever-growing feature models [41, 60, 94], one is therefore interested in reducing the runtime required for analyses. We propose to employ knowledge compilation to reduce the computational effort of multiple queries [24, 26, 56, 68]. Hereby, the original formula is translated to a target format in an offline phase once. If the compilation is successful, the resulting format can be reused for online computations (e.g., SAT or #SAT queries) with typically polynomial-time complexity with respect to the size of the knowledge-compilation artifact [26].

The deterministic decomposable negation normal form (d-DNNF) enables computing the number of satisfying assignments in linear time with respect to its number of nodes [22]. Compilation to d-DNNF is similarly complex as solving a #SAT problem as the d-DNNF is constructed from the trace of one #SAT invocation with available tools [24, 56, 68]. In recent work [94], we showed that d-DNNF compilers [24, 40, 56, 68, 77] are generally faster for feature models than other knowledge compilers that enable model counting. For instance, compiling to d-DNNF is substantially faster than compilation to binary decision diagrams [39, 91, 92]. Thus, d-DNNFs are a promising target format for feature-model counting.

While the compilation to d-DNNF is well researched [24, 40, 56, 68, 77], reusing a given d-DNNF is only explored sparsely [23, 87]. In particular, reusing a d-DNNF for multiple counting queries has not been evaluated before in software engineering or any other domain. Some publications used d-DNNF compilers for counting as black-box #SAT solvers *without* reusing the d-DNNF, essentially compiling a new d-DNNF for every query [53, 94]. Further, Sharma et al. [87] traverse d-DNNFs to derive satisfying assignments for uniform random sampling.

In this work, we propose to *reuse* d-DNNFs for repetitive counting queries which we apply for computing number of valid configurations that (1) contain a certain feature or (2) conform to a partial configuration (i.e., include and/or exclude sets of features). In particular, we compile a given feature model to d-DNNF with a one time effort using existing compilers [24, 56, 68] and then reuse the d-DNNF for numerous counting-based computations. We showcase the vast improvement of our approach over the state-of-the-art with a large-scale empirical evaluation. Thereby, our work focuses on the efficient reuse of d-DNNFs with the following contributions:

First, we present algorithms and optimizations for repetitive counting queries on a given d-DNNF (cf. Section 4). The basic algorithms are adopted from ideas of Darwiche for counting on d-DNNFs [23]. Still, Darwiche's ideas were not followed up further with an implementation or empirical evaluation yet. Further, we contribute a set of novel optimizations which vastly reduce the number of required traversals through the d-DNNF and also accelerate single traversals.

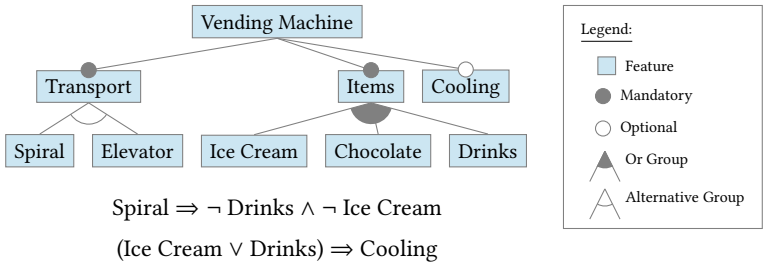



Fig. 1. Feature Model Example

Second, we provide an open-source tool `ddnnife`[1] including all presented algorithms and optimizations (cf. Section 5). Our d-DNNF reasoner `ddnnife` takes a d-DNNF as input supporting all formats used by the popular compilers [24, 56, 68]. Given the d-DNNF, `ddnnife` allows the user to perform counting queries.

Third, we perform a large-scale empirical evaluation comparing the reuse of d-DNNFs with state-of-the-art #SAT solvers on 61 industrial feature models (cf. Section 6). Hereby, we compare our tool `ddnnife` paired with each available d-DNNF compiler to state-of-the-art solutions. The required data, tools, and framework are publicly available for replicability and reproducibility.[2]

## 2 MOTIVATING EXAMPLE

In this section, we show use cases of analyses dependent on multiple counting queries on a small example. Figure 1 shows a feature model representing a simplified vending machine. Each vending machine requires at least one `Item` (denoted as or group) and exactly one `Transport`-system (denoted as alternative group). Furthermore, `Drinks` and `Ice Cream` cannot be sold with a `Spiral`-system, which just drops the items, and `Ice Cream` and `Lemonade` require `Cooling` as denoted by the cross-tree constraints below the tree.

Computing the number of valid configurations enables various analyses. In recent work [92], we presented 18 applications dependent on computing the number of valid configurations (1) collected from the literature [6, 20, 36–38, 53, 93] and (2) inspired from industry collaborations. Those applications are relevant for detecting errors and smells, testing, maintainability, supporting development, economical estimations, and user guidance. 11 of the 18 applications depend on multiple #SAT computations for the same feature model [92].

As an example, the number of valid configurations containing a specific feature can be used to quantify the impact of an error [53, 93]. It may be sensible to address an issue that appears in most valid configurations before an issue appearing in only a small fraction of valid configurations. This is particularly relevant if the configurations used in field are unknown as for the Linux kernel [73]. In our example, an error in feature `Elevator` has an impact on eight valid configurations while an error in feature `Spiral` impacts only two valid configurations.

Feature-model counting can also be used to support users during the interactive configuration of a product line [17]. Here, the number of remaining valid configurations can be used as an indicator for the impact of a user selection. Suppose a user already selected `Chocolate` and thinks about whether they want to select or deselect `Cooling`. When selecting `Cooling`, there are still five configurations possible. Otherwise, only two are left which may be an incentive for selecting `Cooling`. Such information may support the entire configurations process requiring model counts for each remaining feature every time the user makes a selection. Here, response time is critical for the usability.

[1] https://github.com/SoftVarE-Group/d-dnnf-reasoner
[2] https://github.com/SoftVarE-Group/exploiting-ddnnfs-eval/tree/March2023 (Final version: Zenodo)

While both presented example analyses require only a few counting queries for our running example, such analyses often require many queries on large feature models in practice. For instance, homogeneity [30, 37, 92, 93], which describes the similarity of configurations induced by the feature model, requires a counting query for every single feature. Especially, when dealing with large industrial feature models containing thousands of features, performing such analyses may induce large runtimes. For each of the 11 applications requiring multiple #SAT invocations, it may be beneficial to reuse d-DNNFs over multiple computations [92, 93].

## 3 BACKGROUND

In this section, we provide the background required for the following sections. First, we introduce feature models and feature-model counting. Second, we explain d-DNNFs and how to use their structure for model counting.

### 3.1 Feature-Model Counting

For a given product line, a feature model is typically used to specify the set of valid configurations [9]. Formally, we define feature models as tuple $FM = (FT, CT)$ with a set of features $FT$ and constraints $CT$ between these features. The set of constraints $CT$ includes both hierarchical constraints that describe the parent-child relationship, and cross-tree constraints. For the sake of simplicity, we consider the constraints to be a conjunction of *propositional* formulas (i.e., each constraint in $CT$ has to be satisfied to satisfy the feature model). We refer to the resulting formula as $F_{FM} = \bigwedge_{c \in CT} c$. It is well-known that each hierarchical constraint (i.e., alternative, or, mandatory, and optional) can be translated to a propositional formula [9, 21].

A configuration $C = (I, E)$ consists of two sets of features which describe features included ($I \subseteq FT$) in the configuration and excluded ($E \subseteq FT$) in the configuration. A feature cannot be included and excluded in the same configuration (i.e., $I \cap E = \emptyset$). If all features are included or excluded (i.e., $I \cup E = FT$), the configuration is complete. Otherwise, the configuration is partial [9]. We consider a configuration $C = (I, E)$ to be valid, iff it is complete and satisfies all constraints $CT$ imposed by the feature model [9]. We refer to the set of all valid configurations induced by $FM$ as $VC_{FM}$.

Feature-model counting refers to computing the number of valid configurations for a given feature model. In previous work [93], we introduced three types of feature-model counting that compute the number of valid configurations (1) of the entire feature model, (2) that include a certain feature, and (3) that are induced by a partial configuration (i.e., include some and exclude other features). These three types cover all the 21 applications that we identified.

**Definition 1** (Cardinality of a Feature Model)**.** The cardinality $\#FM$ of a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations $VC_{FM}$ (i.e., $\#FM = |VC_{FM}|$).

**Definition 2** (Cardinality of a Feature)**.** The cardinality $\#f$ of a feature $f$ in a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations that include $f$ (i.e., $\#f = |\{C = (I, E) \mid C \in VC_{FM}, f \in I\}|$).

**Definition 3** (Cardinality of a Partial Configuration)**.** The cardinality $\#C_p$ of a partial configuration $C_p = (I_p, E_p)$ given a feature model $FM = (FT, CT)$ corresponds to the cardinality of the set of valid configurations that include each feature $i \in I_p$ and exclude each feature $e \in E_p$ (i.e., $\#C_p = \left|\{C = (I, E) \mid C \in VC_{FM}, I_p \subseteq I, E_p \subseteq E\}\right|$).

Many feature-model analyses are reduced to solving SAT or #SAT problems on a propositional formula representing the feature model [9, 36, 53, 57, 66, 97, 102]. For example, a feature model $FM = (FT, CT)$ is void (i.e., induces no valid configuration [9]) iff the formula $F_{FM}$ is not satisfiable. The three types of cardinality defined above can be reduced to #SAT invocations [36, 93]. The cardinality of a feature model $\#FM$ is equivalent to the number of satisfying assignments of $F_{FM}$ (i.e., $\#\text{SAT}(F_{FM})$). The cardinality of a feature $\#f$ can be computed

**Algorithm 1** Model Counting via d-DNNF Traversal

```
1:  procedure #SAT(ddnnf)
2:      return #NODE(ddnnf.root)
3:  procedure #NODE(node)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c)
8:      else if node is Literal then
9:          return 1
10:     else if node is ⊤ then
11:         return 1
12:     else if node is ⊥ then
13:         return 0
```

via #SAT($F_{FM} \wedge f$). Analogously, the cardinality $\#C_p$ of a partial configuration $C_p = (I, E)$ can be reduced to #SAT($F_{FM} \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e$).

## 3.2 Model Counting with d-DNNFs

Typically, tools based on propositional logic, such as SAT and #SAT solvers, rely on normal forms as representation for input formulas. The most commonly used format is conjunctive normal form (CNF) [8, 14, 16, 24, 25, 45, 49, 56, 68, 78, 86, 88, 98]. However, CNFs do not support model counting in polynomial time [26, 99].

In contrast, the *d-DNNF* (*d*eterministic *D*ecomposable *N*egation *N*ormal *F*orm) format supports linear-time model counting with respect to the number of d-DNNF nodes [24, 26, 68]. The language d-DNNF is a subset of negation normal form (NNF) for which determinism and decomposability holds [22]. Often, d-DNNFs are further restricted to be *smooth* for simplifying and accelerating algorithms on a d-DNNF [23, 25, 26, 68]. All those terms are defined and exemplified below.

**Definition 4** (Negation Normal Form [22])**.** A propositional formula $F$ is in *negation normal form* iff $\wedge$, $\vee$, $\neg$, $\top$, and $\bot$ are the only logical operators and negations ($\neg$) only appear directly in front of literals, $\top$, and $\bot$.

Algorithm 1 shows how to compute the number of satisfying assignments for a given d-DNNF. Starting from the root, the algorithm traverses the d-DNNF and applies rules depending on the node type (i.e., conjunction, disjunction, and literals) exploiting the properties decomposability, determinism, and smoothness. In the following, we explain these three properties of d-DNNFs and their impact on Algorithm 1.

**Definition 5** (Decomposability [24])**.** A propositional formula $F$ is *decomposable* iff for each conjunction $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ the sets of variables of each conjunct $C_i$ are disjunct (i.e., $\forall\, i, j : i \neq j : vars(C_i) \cap vars(C_j) = \emptyset$).

With decomposability, the number of satisfying assignments of a conjunction is the product of its children's numbers of satisfying assignments, as seen in Lines 2-3 of Algorithm 1. If a formula $F$ is decomposable, the conjuncts of each conjunction share no variables. Thus, we can combine the satisfying assignments pairwise (i.e., building the cross-product) for each of the conjuncts. Due to the pairwise combination, the size of the resulting set (i.e., the number of satisfying assignments) is the product of sizes for the respective conjuncts.

Consider the example formula $F = (A \vee B) \wedge (C \vee \neg C)$. The left part $A \vee B$ induces the three satisfying assignments $\{A, B\}, \{A, \neg B\}, \{\neg A, B\}$. The right part $C \vee \neg C$ induces the two satisfying assignments $\{C\}, \{\neg C\}$. Pairwise combining both sets results in $3 \cdot 2 = 6$ satisfying assignments for $F$ (i.e., $\{A, B, C\}$, $\{A, B, \neg C\}$, $\{A, \neg B, C\}$,

$\{A, \neg B, \neg C\}$, $\{\neg A, B, C\}$, $\{\neg A, B, \neg C\}$). Without decomposability, the described procedure may lead to faulty results as the combined assignments may be conflicting or not disjunct leading to faulty results when counting.

**Definition 6** (Determinism [24])**.** A propositional formula $F$ is *deterministic* iff for each disjunction $D = D_1 \vee D_2 \vee \ldots \vee D_m$, the disjuncts share no common satisfying assignments (i.e., $\forall\, i, j : i \neq j : D_i \wedge D_j = \bot$).

With determinism, the number of satisfying assignments of a disjunction is the sum of its children's numbers as seen in Lines 4–5 of Algorithm 1. For a disjunction, each assignment that satisfies one of the disjuncts also satisfies the disjunction. Therefore, the set of satisfying assignments is the union over the set of the conjuncts. For a deterministic disjunction, there are no solutions shared between disjuncts. Thus, when conjoining the set of satisfying assignments of all disjuncts there are no duplicate entries.

Consider the example formula $F = (A \wedge B) \vee (\neg B)$. $(A \wedge B)$ and $(\neg B)$ induce $A = \{\{A, B\}\}$ and $B = \{\{A, \neg B\}, \{\neg A, \neg B\}\}$ as satisfying assignments, respectively. The set of satisfying assignments for $F$ is the union $A \cup B$ of both sets. As $A$ and $B$ are disjunct by definition, the number of satisfying assignments is the sum of the number of satisfying assignments for its children. Without determinism, the described procedure may lead to faulty results due to duplicate satisfying assignments.

**Definition 7** (Smoothness [22])**.** A propositional formula $F$ is *smooth* iff for each disjunction $D = D_1 \vee D_2 \vee \ldots \vee D_m$, the disjuncts have the same set of variables ($\forall i, j : vars(D_i) = vars(D_j)$).

Smoothness is often enforced for disjunctions to remove the overhead of tracking variables when traversing the d-DNNF [23, 24, 26, 68]. For a smooth disjunction, the sets of variables are equal for every disjunct by definition. Without smoothness, if one of the disjuncts $D_i$ contains a variable $v$ that does not appear in another disjunct $D_j$, $v$ can be freely assigned (i.e., either $\top$ or $\bot$) in the context of $D_j$. In this case, the number of satisfying assignments for $D_j$ needs to be multiplied by a factor of two for each missing variable. Consequently, without smoothness, Line 5 in Algorithm 1 would require an adaptation to cope with free variables. As we presume smoothness, the algorithm does not need to keep track of the set of variables. According to Darwiche [22], every d-DNNF can be smoothed in polynomial time with respect to its number of nodes and variables. In this work, we consider a d-DNNF to be smooth if not stated otherwise.

Due to the presented properties, namely decomposability, determinism, and smoothness, the d-DNNF can be traversed with Algorithm 1 to compute the number of satisfying assignments of a propositional formula [23]. Note that each literal induces exactly one satisfying assignment (i.e., setting the respective variable to $\top$ for $f$ or $\bot$ for $\neg f$) as seen in Lines 6–7 of Algorithm 1. Further, $\top$ induces one satisfying assignment and $\bot$ zero.

The cardinality $\#FM$ of a feature model $FM = (FT, CT)$ is equivalent to the number of overall satisfying assignments $\#F_{FM}$. Hence, the algorithm can be directly used to compute the number of valid configurations for the entire feature model using a d-DNNF representing $F_{FM}$ [87, 91]. In Section 4, we adapt the algorithm to compute the cardinalities of features and partial configurations.

## 4 REUSING D-DNNFS FOR CARDINALITY OF FEATURES AND PARTIAL CONFIGURATIONS

In this work, we use the benefits of knowledge compilation to d-DNNF for feature-model counting. In particular, we focus on repetitive queries as required for cardinality of features and partial configurations. For a given feature model, we compile it to d-DNNF in a one-time offline phase using existing compilers [24, 56, 68]. Afterwards, we reuse the compiled d-DNNF by reducing the two types of cardinalities queries on this d-DNNF. Those queries can be computed in linear time with respect to the number of nodes in the d-DNNF. The respective d-DNNF queries shown in Section 4.1 are inspired from ideas of an algorithm introduced by Darwiche [23] but have never been implemented nor evaluated. Further, we propose several optimizations to those queries in Section 4.2.

### 4.1 d-DNNF Queries for Features and Partial Configurations

Algorithm 2 and Algorithm 3 respectively depict the computation for cardinality of features and partial configurations via d-DNNF traversal. Respectively, the procedures `#FEATURE` and `#CONFIG` compute the cardinality of one feature or configuration by recursively applying rules based on the node type starting from the root node of the d-DNNF. Hereby, both algorithms adapt Algorithm 1 by performing different computations for the literals. Note that Lines 3–11 are exactly equal for both algorithms. Figure 2 shows an example d-DNNF representing the formula $A \wedge ((B \wedge \neg C) \vee (\neg B \wedge C)) \wedge (D \vee \neg D)$[3] as direct acyclic graph.

Each node corresponds to a logical operator or literal. The three numbers next to the nodes represent the number of satisfying assignments of that node for example queries of the three types of cardinalities. In Figure 2, the black numbers refer to traversing the d-DNNF to compute the overall number of satisfying assignments (equivalently: valid configurations) as seen in Algorithm 1. The result in root node of the d-DNNF in Figure 2 indicates a cardinality of $\#F_{FM} = 4$.

**Algorithm 2** Cardinality of Features

```
1:  procedure #FEATURE(ddnnf, feature)
2:      return #NODE_FEAT(ddnnf.root, feature)
3:  procedure #NODE_FEAT(node, feature)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c, feature)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c, feature)
8:      else if node is ⊤ then
9:          return 1
10:     else if node is ⊥ then
11:         return 0
12:     else if node is Literal then
13:         return #LITERAL_FEAT(node, feature)
14: procedure #LITERAL_FEAT(literal, feature)
15:     if literal.variable == feature then
16:         if literal.positive then return 1
17:         else return 0
18:     else return 1
19:
20:
21:
22:
```

**Algorithm 3** Cardinality of Partial Configurations

```
1:  procedure #CONFIG(ddnnf, config)
2:      return #NODE_CONFIG(ddnnf.root, config)
3:  procedure #NODE_CONFIG(node, config)
4:      if node is And then
5:          return ∏_{c∈children} #NODE(c, config)
6:      else if node is Or then
7:          return ∑_{c∈children} #NODE(c, config)
8:      else if node is ⊤ then
9:          return 1
10:     else if node is ⊥ then
11:         return 0
12:     else if node is Literal then
13:         return #LITERAL_PC(node, config)
14: procedure #LITERAL_PC(literal, config)
15:     if literal.variable ∈ config.included then
16:         if literal.positive then return 1
17:         else return 0
18:     else if literal.variable ∈ config.excluded then
19:         if literal.positive then return 0
20:         else return 1
21:     else
22:         return 1
```

*Features.* To compute the cardinality $\#f$ of a feature $f \in FT$ with $FM = (FT, CT)$, we need to exclude valid configurations $C = (I, E)$ with $C \in VC$ that do not include $f$ (i.e., $f \notin I$). Therefore, when traversing the d-DNNF to compute $\#f$, we discard each satisfying assignment that falsifies $f$ by setting negative literals $\neg f$ to zero. In Figure 2, the blue numbers refer to the cardinality of $B$. The literal $\neg B$ is initialized with a value of zero. Positive literals of $B$ and other literals are initialized with a value of one. The handling of literals for the cardinality of a feature is described in lines 8–12 of Algorithm 2. Using the adapted literal values, the number of satisfying

[3]The tautology $(D \vee \neg D)$ is included to have another simple subformula that is smooth, deterministic, and decomposable. In practice, such structures are often part of d-DNNFs to ensure smoothness (cf. Section 5.1) or to include fully optional variables in the d-DNNF.

assignments including $B$ can be computed by traversing the d-DNNF which results in a cardinality $\#f = 2$ as seen in the root of Figure 2.

*Partial Configurations.* For the cardinality $\#C_p$ of a partial configuration $C_p = (I_p, E_p)$, we exclude assignments that assign false to an included feature $i \in I_p$ or true to an excluded feature $e \in E_p$. In Figure 2, the orange numbers indicate the cardinality for the partial configuration $\{\neg C, D\}$. For each included feature (here: $D$), positive and negative literals are initialized with one and zero, respectively. For each excluded feature (here: $C$), positive and negative literals are initialized with zero and one, respectively. Literals that correspond to features that are neither included nor excluded are always initialized with a value of one. The handling of literals for the cardinality of a partial configuration is specified in Lines 8–16 of Algorithm 3. After initializing the values for literal nodes, the d-DNNF can be traversed to compute the cardinality $\#C_p = 1$ as seen in the root of Figure 2.

*Correctness.* To illustrate the correctness of our proposed Algorithm 2 and Algorithm 3, we presume that the original algorithm (Algorithm 1) is correct [23, 26]. The correctness of Algorithm 1 depends on the three properties decomposability, determinism, and smoothness. When applying the assumptions for features and partial configurations, we essentially *condition* the formula for each variable of interest. Alternatively stated, for a decision $d$ we replace each conflicting literal $\neg d$ with $\bot$ (false) and literals $d$ with $\top$ (true). Hence, the adapted formula represents the subset of satisfying assignments that adheres to the assumptions. For the correctness of Algorithm 2 and Algorithm 3, it remains to show that the adapted formula is still (1) decomposable, (2) deterministic, and (3) smooth after applying the assumptions. First, decomposability is never violated as we only remove variables. It follows that the sets of variables for each conjunct of an $\wedge$-node are still disjunct afterwards. Second, each deterministic $\vee$-node is still deterministic. With decision propagation, we never add new satisfying assignments but only remove them. Hence, two sets of satisfying assignments that were disjunct prior to applying the assumptions are still disjunct. Third, for a smooth $\vee$-node, we remove the same set of variables for every child node. Hence, if the set of variables were equal before applying the assumptions, the set of variables stay equal after. Overall, Algorithm 2 and Algorithm 3 yield correct results since the adapted formula (1) represents the set of satisfying assignments conforming to the query and (2) is still a smooth d-DNNF.

*Summary.* With the presented algorithms, d-DNNFs can be reused to compute results for every application based on the cardinality of features and partial configurations [92]. Each described algorithm is an online computation on a given d-DNNF and has a linear runtime with respect to the number of d-DNNF nodes.

## 4.2 Optimizations

For the algorithms presented in Section 4, every node of a given d-DNNF needs to be traversed for each query. Even though the complexity per query is linear in the number of nodes, analyses in practice may induce a high number of queries on d-DNNFs which may be exponentially larger than the CNF and contain millions of nodes

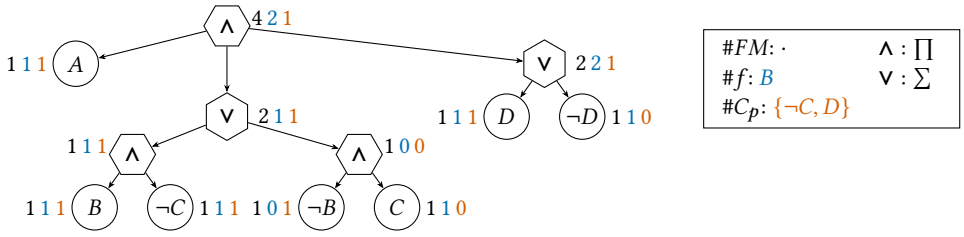



Fig. 2. Computing Cardinalities on a d-DNNF

in practice. Further, the algorithm requires processing large numbers with up to thousands of digits in each step [94]. We propose several optimizations to accelerate d-DNNF traversals to answer queries and reduce the number of required traversals.

*Partial Traversals.* With partial traversals, we aim to accelerate the d-DNNF traversals by reusing results of the overall model count (#$FM$ in Figure 2). If we compute the number of satisfying assignments under a partial assignment of literals (e.g., for the cardinality of a feature), we only have to recompute the value of nodes that are influenced by a corresponding literal. Hence, for each query, we first traverse the nodes of a d-DNNF to identify nodes that are part of a direct path from a corresponding literal to the root and mark these nodes. Afterwards, we traverse the d-DNNF and only recompute the count of marked nodes. To reuse values that do not change due to a given partial assignment, we cache the original value (i.e., without partial assignment) for every node. Figure 3 shows the partial traversal (blue path) of computing the cardinality of $B$ for our running example. The traversal only requires adapting the values from a direct path between $\neg B$ and the root $\wedge$. Note that for every unaffected/uncolored node, the value for the cardinality of $B$ and the original value is equal. This also applies to the node $B$ which still induces exactly one satisfying assignment when conditioned with the query $B$. Hence, only four out of the twelve overall nodes need to re-evaluated with our partial traversal.

*Iterative Traversal.* In practice, d-DNNF nodes often have several parents [56]. Hence, a recursive descent could either redundantly traverse some subtrees multiple times or require overhead to prevent this. To simplify reusing values and overall reduce the computational effort when traversing the d-DNNF, we use an iterative approach instead of a recursive one. The nodes are stored in a list such that a child is always traversed before its parents. Thus, when processing a node, the values of its children are already available and do not require any recursive descent. Figure 4 shows the list representation we use for the d-DNNF seen on the left side. From left to right, the list representation shows the type of the node (i.e., $\wedge$), the original count, the adapted count, and pointers to its children. To evaluate a query (here: #$A$) on the d-DNNF, the list is traversed iteratively starting from the first entry in the list. In this case, we start with the node representing the literal $\neg A$. When reaching one of the nodes depending on the values of other nodes, all required values are already computed. For instance, when reaching the first OR node, we just look up the values for $C$ and $\neg C$.

For partial traversals, we ensure to visit only marked nodes. The dashed blue lines indicate which nodes were marked during the first phase for the query #$A$. Afterwards, the algorithm recomputes the value for each of the three marked nodes. The result for the query #$A = 2$ is then saved in the adapted value of the root node.

*Partial Calculations.* When computing the value of a node during a partial traversal, it is possible only for a small subset of its child nodes to be changed. In this case, not recomputing the entire node but adapting the cached value may accelerate the d-DNNF traversal. For instance, given a $\wedge$-node with ten children for which only one child $c'$ changed, we do not build the product of the ten children. Instead, we adapt the original value

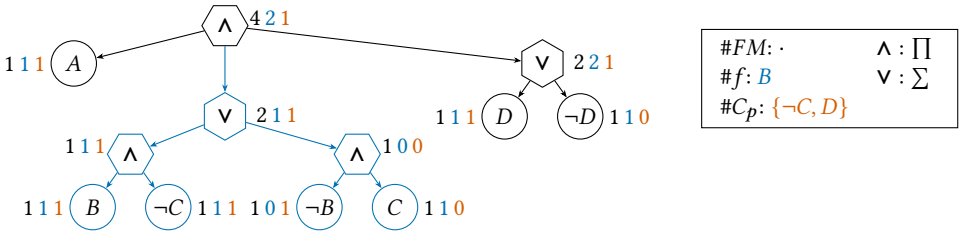



Fig. 3. Partial Traversal on a d-DNNF for the Cardinality of Feature $B$

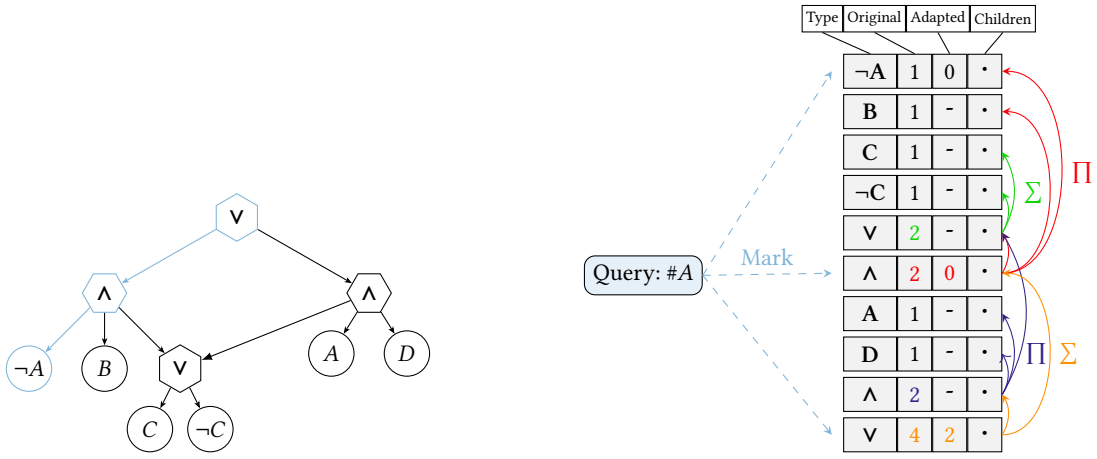



Fig. 4. List Format for Iterative Traversal With Sharing

based on the changed values. If more than half of the children changed, we recompute the value as shown in Algorithm 1. Otherwise, we divide the cached value by the old value of $c_{old}$ and then multiply it with the new value for every changed child. Analogously, we subtract the cached value by the old value of $c_{old}$ and then add the new value for $\vee$ nodes. The partial calculation requires two arithmetic operations per changed node. For instance, recomputing an $\wedge$-node requires one multiplication and one division. Hence, we only apply partial calculation if less than half of the children values changed.

*Core and Dead Features.* The knowledge about core and dead features can be used to reduce the number of required queries. Core and dead features appear in all and no valid configurations, respectively [9]. Hence, the cardinality of core and dead features can be directly derived without traversing the d-DNNF again as seen in Equation 1. For a core feature, the number of satisfying assignments is always equal to the number of overall satisfying assignments. The cardinality of a dead feature is always zero. Further, we can also use core and dead features to compute the cardinality of a partial configuration without traversal in some cases. Each time a dead feature is included or a core feature is excluded in a partial configuration, this configuration is unsatisfiable and does not need to be evaluated with a d-DNNF traversal. Further, included core features and excluded dead features can be disregarded for the marking phase of partial traversals.

We propose to compute core and dead features in constant time per feature by exploiting the smoothness of a d-DNNF. With our proposal, we can identify all core and dead features with *one* traversal through the literal nodes using the following properties. A feature $f$ is core iff only positive literals $f$ appear in the d-DNNF. Analogously, a feature $f$ is dead iff only negative literals $\neg f$ appear. In our running example Figure 2, the feature $A$ is core as only positive literals appear in the smooth d-DNNF.

$$\#f = \begin{cases} \#FM & \text{if } f \text{ is core} \\ 0 & \text{if } f \text{ is dead} \\ \#f & else \end{cases} \tag{1}$$

In the following, we provide a short intuition on the correctness of our approach that detects core and dead features. Assuming that our algorithms for the cardinality of partial configurations are correct, the correctness of our proposal using core/dead features is directly implied as explained in the following. We start the explanations

with core features. If only positive literals of a variable $v$ appear, applying Algorithm 2 for $\#v$ (i.e., cardinality of $v$) always produces the same results as the d-DNNF under no assumptions. The only case resulting in a change would be a negative literal of $v$ which conflicts our assumption of only having positive literals of $v$. Hence, the number of satisfying assignments $v$ appears in is equal to the number of overall satisfying assignments making $v$ a core feature. This works analogously for dead features. If only negative literals of a variable $w$ appear, evaluating $\#(\neg w)$ with Algorithm 2 always yields the same result as for the d-DNNF without assumptions. Thus, there are no satisfying assignments including $w$ in this case as otherwise the results would differ.

*Partial Derivatives.* For the cardinality of features, we can apply partial derivatives on d-DNNFs as presented by Darwiche [23] instead of querying the d-DNNF for each feature. A partial derivative indicates the impact on the overall result of changing one of the variables under the assumption that every other variable is fixed [23]. As cardinality of features relies on flipping single literals for each computation, we can employ partial derivatives as optimization. The idea is to consider the d-DNNF as computation graph with each $\wedge$ being a multiplication and each $\vee$ being an addition. Here, we can annotate the different nodes with partial derivatives that then enable to compute the cardinality of a feature in constant time. Partial derivatives are not generally applicable for partial configurations, since flipping multiple literals is typically required.

The partial derivatives can be computed by traversing the d-DNNF twice. In the first traversal, each node is annotated with its model count as done in Algorithm 1. In the second traversal, these counts are used to derive the partial derivatives $\frac{\delta F}{\delta n}$ of node $n$ in the context of formula $F$ as shown in Equation 2. Here, $P(n)$ is the set of parents of $n$, $C(p)$ the set of children of $p$, and $\#c$ the model count of the node $c$.

$$\frac{\delta F}{\delta n} = \begin{cases} 1 & \text{n is root} \\ \sum_{p \in P(n)} PD(n,p) & \text{else} \end{cases} \tag{2}$$

$$PD(n,p) = \begin{cases} \frac{\delta F}{\delta p} & \text{p is } \vee \\ \frac{\delta F}{\delta p} \cdot \prod_{c \in C(p) \setminus \{n\}} \#c & \text{p is } \wedge \end{cases} \tag{3}$$

Figure 5 shows the computation graph for Figure 2 annotated with the partial derivatives. Given the partial derivatives, the cardinality of each feature can be computed with Equation 4. Here, $\#FM$ is the original count, $\Delta(f)$ is the change of the variable (e.g., -1 for setting $f$ from true to false), and $\frac{\delta F}{\delta f}$ is the partial derivative of $f$. For instance, setting $A$ to false reduces the overall model count by 4 as indicated by the orange number in Figure 5.

$$\#f = \#FM + \Delta(f) \cdot \frac{\delta F}{\delta f} \tag{4}$$

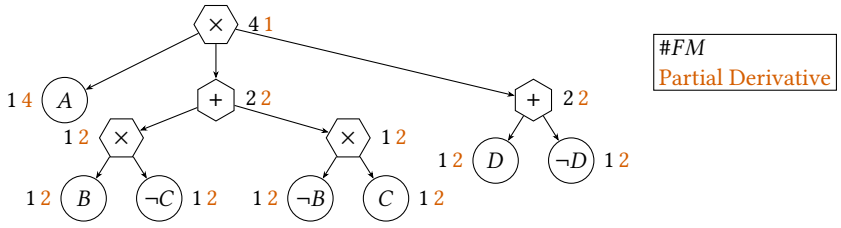



Fig. 5. Computation Graph with Partial Derivatives for d-DNNF Shown in Figure 2

*Summary.* Each of the five presented optimizations reduces either the number of required queries or reduces effort for a single traversal. Both of those impacts potentially improve the efficiency of reusing d-DNNFs for computing the cardinality of features or partial configurations. In Section 6, we examine the general performance of our approach and also take a closer look on the effect of specific optimizations.

## 5 TOOL SUPPORT: DDNNIFE

The presented algorithms and optimizations are implemented in our open-source d-DNNF reasoner `ddnnife`, the Swiss *knife* for reusing d-DNNFs. `ddnnife` is implemented in Rust[4] as Rust generally compiles to efficient target code and has great support for improving memory safety. The source code and example input data is available at GitHub[5] under LGPLv3[6] license which allows usage of `ddnnife` with only few restrictions even in commercial settings. We use `ddnnife`, amongst other tools, to evaluate the advantages of reusing d-DNNFs in Section 6.

### 5.1 Parsing d-DNNFs

*Input Format.* `ddnnife` supports the standard format set by `c2d` [24] and the format introduced by `d4` [56]. Supporting both formats, `ddnnife` is able to parse d-DNNFs from each popular d-DNNF compiler, namely `c2d` [24], `dSharp` [68] (also uses `c2d` format), and `d4` [56]. In addition, `ddnnife` supports CNFs in DIMACS-format [85] as input. Here, `d4` is internally used to translate the CNF to d-DNNF.

Listing 1 shows our running example Figure 2 in the `c2d` format.[7] The first line `nnf v e n` indicates the number of nodes (`v`), the number of edges (`e`), and the number of variables (`n`). Each remaining line describes one node in the d-DNNF. `L 1` introduces a positive literal of the variable `1`. An and node is described by the line `A n i j k` with `n` children and `i`, `j`, `k` being indices of the lines holding child nodes. Note that the first line containing a node has the index zero. Or nodes come with an additional parameter that indicates a decision variable (i.e., a variable that is always true in the left and false in the right child). `O d n i j k` describes an or node with the decision variable `d`, `n` child nodes with the indices `i`, `j`, and `k`. If `d` is `0`, there is no decision variable.

Listing 2 shows our running example Figure 2 in the `d4` format.[8] Here, each line either describes a node or an edge between nodes and is terminated with a `0`. Lines describing nodes always first specify the type of node, namely or (o), and (a), true (t), false (f), and then the index of this node. In contrast to the `c2d`-format, the indices do not refer to the line index but to the index specified for each node. Edges always follow the format `p c x y z 0` with the parent node `p`, the child node `c`, and an arbitrary number of literals `x y z` set to true when following this path. For instance, Line 4 specifies that the node with the index `1` has a child node $\Phi_3$ with the index `3`. Further, the literals `2` and `-3` are conjoined resulting in the subformula $(\Phi_3 \wedge (2 \wedge -3)) \vee \Phi_{rest}$ with $\Phi_{rest}$ being the formula resulting from other edges starting from node `1`.

[4] https://www.rust-lang.org/
[5] https://github.com/SoftVarE-Group/d-dnnf-reasoner
[6] https://www.gnu.org/licenses/lgpl-3.0.de.html
[7] http://reasoning.cs.ucla.edu/c2d/
[8] https://github.com/crillab/d4

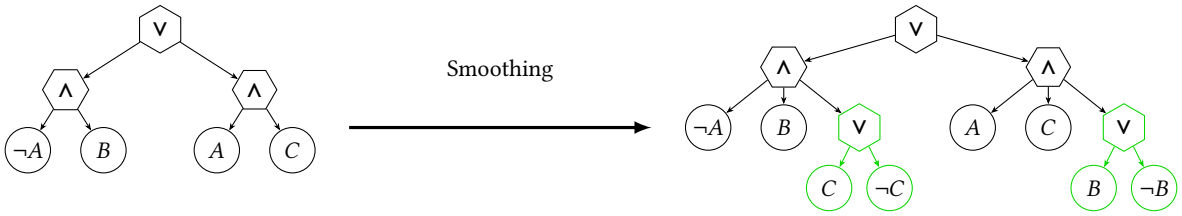



Fig. 6. Smoothing a d-DNNF

Listing 1. c2d Format

```
 1  nnf 11 11 4
 2  L 1 # A
 3  L 2 # B
 4  L -3 # !C
 5  L -2 # !B
 6  L 3 # C
 7  L 4 # D
 8  L -4 # !D
 9  A 2 1 2
10  A 2 3 4
11  O 0 2 7 8
12  O 4 2 5 6
13  A 3 0 9 10
```

Listing 2. d4 Format

```
 1  o 1 0
 2  o 2 0
 3  t 3 0
 4  a 4 0
 5  1 3 2 -3 0
 6  1 3 -2 3 0
 7  2 3 4 0
 8  2 3 -4 0
 9  4 3 1 0
10  4 0 2 0
11  4 0 1 0
```

*Smoothing.* As discussed in Section 3.2, computing the cardinalities on an unsmooth d-DNNF causes computational overhead as the algorithm needs to keep track on the set of variables during the traversal [26]. While `c2d` and `dSharp` provide a functionality to produce smooth d-DNNFs, `d4` does not guarantee smoothness. Consequently, the procedure described below is mainly valuable for d-DNNFs compiled by `d4`. Note that similar procedures are used in `dSharp` and `c2d` but are only sparsely documented [24, 69].

To ensure smoothness, we add designated nodes for each unsmooth $\vee$-node if needed, as seen in Figure 6 in a preprocessing step. For each missing variable $v \in \mathit{missingVariables}_{\Phi}$ in a child subtree $\Phi$ of the $\vee$-node, we add a disjunction $v \vee \neg v$ resulting in the following smooth disjunction on the right side.

$$\bigvee_{\Phi \in children} \Phi \quad \rightarrow \quad \bigvee_{\Phi \in children} (\Phi \wedge \bigwedge_{v \in missingVariables_{\Phi}} v \vee \neg v)$$

The added subformulas are tautologies as $v \vee \neg v \equiv \top$ holds for every variable $v$. Hence, when adding smoothing nodes to a subformula $\Phi$, the result $\Phi \wedge \top$ equivalent to $\Phi$. In the example in Figure 6, the left subtree does not include the variable $C$. Analogously, the right subtree does not include $B$. Hence, the set of variables of the subtrees are not equal, which violates smoothness. We conjunct a smoothing node $C \vee \neg C$ and $B \vee \neg B$ to the subtrees, respectively. After the smoothing step, both sides of the root $\vee$-node include the same set of variables $\{A, B, C\}$ without changing the semantics of the formula.

*Pre-Processing.* Prior to performing queries for features or partial configurations, `ddnnife` performs some pre-processing steps to simplify the computation for those queries. First, the given d-DNNF is smoothed as

described above. Second, connections from child to parent node are added to simplify the marking algorithm for partial traversals. Third, pointers to the indices of literal nodes are saved to be used as starting point for marking. Fourth, core and dead features are identified by examining if a variable only appears as positive (core) or negative (dead) literal, respectively. Fifth, the model count for each d-DNNF node is computed under no assumptions. Those values are then re-used for partial traversals. After the different the pre-processing steps, the resulting representation of the d-DNNF is then stored and used for querying.

## 5.2 Implementation of Algorithms

`ddnnife` supports the algorithms for computing the cardinality of feature models, features, and partial configurations presented in Section 4.1. By default, we apply partial *derivatives* for cardinality of features and partial *traversals* for partial configurations. As the parsing step ensures that each d-DNNF is smooth, the algorithms do not keep track of variables during the computation.

*Optimizations.* `ddnnife` includes all optimizations presented in Section 4.2. However, some optimizations are slightly adapted according to observations when applying `ddnnife` to d-DNNFs representing feature models.

Partial calculation is only applied to $\wedge$-nodes, due to the following two reasons. First, multiplications are more computationally expensive than addition resulting in smaller runtime profits when reducing the number of additions. Second, in practice, $\vee$-nodes typically have exactly two child nodes due to the construction of d-DNNFs with `c2d`, `dSharp`, and `d4`. For exactly two children, the partial-calculation optimization introduces a small overhead but provides no benefit as still two arithmetic operations are required when only one child changes.

When dealing with large partial configurations (i.e., many included or excluded features), a large portion of the d-DNNF changes and needs to be traversed. Thus, if the size of the partial configurations exceeds a certain threshold, the overhead of marking nodes to recompute is no longer worth it. In this case, we do not use partial traversals but re-evaluate the whole d-DNNF.

## 5.3 Tool Usage

Prior to employing different analyses, `ddnnife` always demands a d-DNNF, either in `c2d` (also used by `dSharp`) or `d4` format, as input. In this section, we shortly explain the different modes of usage for `ddnnife`, namely a one-time computation for a given set of queries and streaming mode for interactive usage. For more details on syntax and technical features, we refer to the repository.[9]

*One-Time Computation.* When invoking d-DNNF as one-time computation, the user has four options to denote queries: (1) compute cardinality of a feature, (2) cardinality of all features, (3) cardinality of a partial configuration, or (4) a set of different queries given in a text file. After invocation, the tool provides the results either as `.csv` or command line output and then terminates.

*Interactive Mode.* The interactive mode of `ddnnife` can be used for interactive querying if not all queries are known in advance. At the beginning, `ddnnife` loads and pre-processes the d-DNNF in a one-time effort and waits for queries on features or partial configurations. The interactive mode follows a strict protocol to allow users integrating `ddnnife` in non-rust based tools. Nevertheless, `ddnnife` can also be used interactively with human input. After loading, the user can query the d-DNNF (e.g., `count v 1` to count the cardinality of variable 1). For details on the protocol, we also refer to the repository.

*Options.* Besides usage mode and parameters for specifying queries, `ddnnife` provides some additional parameters to control its behavior. Often, pre-processing and in particular smoothing the d-DNNF requires a considerable

[9] https://github.com/SoftVarE-Group/d-dnnf-reasoner

initial effort. Hence, we provide an option to save the smoothed d-DNNF for later usage. Multithreading can also be enabled by the user to answer multiple queries in parallel. In some cases, `d4` omits fully optional features in the d-DNNF. When disregarding those variables, two problems emerge. First, the number of satisfying assignments would be off by two to the power of omitted variables as those can be assigned arbitrarily. Second, `ddnnife` could not compute the cardinality of those features or partial configuration including/excluding one of those as they are not part of the d-DNNF. Hence, we provide a parameter for declaring omitted variables to prevent faulty results.

*Summary.* For a given d-DNNF, `ddnnife` can be used interactively or with a one-time computation to perform repetitive counting queries. With popular compilers, such as `d4` [56], `c2d` [24], and `dSharp` [68], we can translate a feature model to d-DNNF to perform feature-model counting with `ddnnife`. In Section 6, we examine the performance of `ddnnife` and its optimizations when applied to feature models. Hereby, we focus on the mode for one-time computations.

## 6 EVALUATION

In our empirical evaluation, we analyze the benefits of reusing d-DNNFs for computing the cardinality of features and partial configurations compared to the state of the art. The applicability of applying d-DNNFs for feature-model counting depends on the scalability of both the compilation (offline phase) and the reuse of the compiled d-DNNF with a reasoner (online phase). Hence, we consider both phases in our empirical evaluation. We provide a publicly available replication package[10] that includes all evaluated tools, subject systems (except nine confidential systems of our industry partner), and the required tooling for measurements.

### 6.1 Research Questions

With our research questions, we aim to inspect the advantages of reusing d-DNNFs compared to repetitive #SAT calls. First, we check if using d-DNNFs is generally *applicable* by studying the scalability of available d-DNNF compilers (RQ1). Second, we empirically examine the *correctness* of considered tools (RQ2). Third, we evaluate the runtime *improvements* when reusing d-DNNFs compared to the state-of-the-art (RQ3). Fourth, we compare the available options for reusing d-DNNFs to find *promising candidates* for usage in practice (RQ4).

**RQ1** How efficient are d-DNNF compilers when applied to industrial feature models regarding runtime and memory?

To apply our algorithms on industrial feature models, it is necessary to be able to compile the respective formulas to d-DNNF first. With RQ1, we inspect and compare the scalability of d-DNNF compilers regarding runtime and size of the resulting d-DNNFs. We also consider the sizes, as the runtime of `ddnnife` depends on the size of the d-DNNF. While the general performance of the compilers have been considered in other work, the existing evaluations either (1) do not evaluate on feature models [15, 24, 56, 68], (2) use d-DNNF compilers as black-box #SAT solvers [53, 91], or (3) do not consider the size of the d-DNNFs [92]. In particular, the sizes of d-DNNFs representing feature models have not been studied.

**RQ2** Which tools and tool combinations compute correct cardinalities?

Correctness is another vital aspect for applicability and also for sound performance comparisons. To examine the correctness of considered tools and in particular our approach `ddnnife`, we compare computed cardinalities.

**RQ3** How efficient is reusing d-DNNFs for computing cardinalities of features and partial configurations compared to repetitive #SAT calls?

[10] https://zenodo.org/doi/10.5281/zenodo.12699707

After examining the general applicability with RQ1 and RQ2, we inspect the main focus of this work (i.e., advantages of reusing d-DNNFs for feature-model counting) with RQ3. We compare runtimes of d-DNNF-based approaches with the runtimes of repetitive #SAT calls for computing cardinality of features and partial configurations. Hereby, we consider reusing d-DNNFs with and without compilation time for comparison.

**RQ4** Which d-DNNF-based tools perform best for computing cardinalities?

After evaluating the difference between d-DNNF-based approaches and the state-of-the-art in RQ3, we focus on comparing the different d-DNNF-based tools in RQ4. In particular, we aim to provide recommendation on which combinations of compiler and reasoner to use.

## 6.2 Experiment Design

*Subject Systems.* In our empirical evaluation, we only consider industrial feature models and no artificial ones. Artificially generated feature models may result in runtimes that are not representative for the real world as observed previously [5, 39]. With our selection of subject systems, we aim for a wide coverage of industrial feature models from a variety of domains. To this end, we mostly use publicly available feature models used in benchmarks in related work. Table 1 gives an overview on the subject systems considering domain, number of features, number of clauses, and cardinality.

First, we include feature models from a benchmark[11] provided by Knüppel et al. [47] which consists of various CDL[12] models, KConfig[13] models, and an automotive product line. For the 117 CDL feature models, we observed in previous work [91] that they are highly similar considering various metrics, including number of features, number of constraints, and also performance of #SAT solvers. To prevent a bias, we decided to include only three CDL feature models instead of all 117 in our benchmark. Here, we took the feature models with the minimum, median, and maximum number of features. Second, our benchmark includes feature models from the systems software and application domain[14] provided by Oh et al. [76]. Third, we include a version of the BusyBox feature model[15] provided by Pett et al. [80]. Fourth, we consider a feature model[16] extracted from the financial services domain [72, 81]. Fifth, we consider another feature model from the automotive domain published by Kowal et al. [50]. Sixth, we include a database feature model publicly available in the FeatureIDE examples.[17] Sixth, we include nine feature models representing automotive product lines from an industry collaboration, which unfortunately cannot be published. Overall, our benchmark consists of 62 industrial feature models.

*Evaluated Tools.* For the selection of tools, our goal is to compare d-DNNF compilation and reuse to the state-of-the-art of feature-model counting. Currently, the most scalable solution that is generally applicable appears to be translating a feature model to CNF and invoking a #SAT solver as black box [70, 75, 94]. As representative of the state-of-the-art, we include the fastest publicly available off-the-shelf #SAT solvers [16, 88, 94, 98]. For representative results, we aim to include each publicly available d-DNNF compiler [24, 56, 68] (i.e., tools compiling CNF to d-DNNF) and reasoner (i.e., tools using d-DNNF to compute the number of satisfying assignments).

Table 2 provides an overview on the tools we identified. Every listed tool is included in our replication package. In a report of the model counting competition 2020 (MC2020), Fichte et al. [31] list existing tools capable of computing the number of satisfying assignments including d-DNNF compilers. In our empirical evaluation,

[11] https://github.com/AlexanderKnueppel/is-there-a-mismatch
[12] https://ecos.sourceware.org/
[13] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[14] https://github.com/jeho-oh/Smarch
[15] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling
[16] https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines
[17] https://github.com/FeatureIDE/FeatureIDE/tree/v3.11.1/plugins/de.ovgu.featureide.examples/featureide_examples

Table 1. Overview Subject Systems (sorted by Domain and Variables)

| System [Source] | Domain | Variables | Clauses | Cardinality |
|---|---|---|---|---|
| lrzip [75] | Archiver | 20 | 63 | 144 |
| 7z [75] | Archiver | 44 | 210 | 68,640 |
| Automotive03a [confidential] | Automotive | 384 | 1,020 | $5.16 \cdot 10^{10}$ |
| Automotive06a [confidential] | Automotive | 449 | 1,339 | $1.13 \cdot 10^{10}$ |
| Automotive04 [confidential] | Automotive | 531 | 5,961 | $2.45 \cdot 10^{21}$ |
| Automotive07a [confidential] | Automotive | 581 | 2,235 | $4.34 \cdot 10^{15}$ |
| Automotive03 [confidential] | Automotive | 588 | 4,638 | $2.54 \cdot 10^{31}$ |
| Automotive05 [confidential] | Automotive | 1,663 | 45,795 | unknown |
| Automotive01 [50] | Automotive | 2,513 | 10,275 | 5.43+217 |
| Automotive03b [confidential] | Automotive | 4,902 | 9,922 | 13,984 |
| Automotive06b [confidential] | Automotive | 5,571 | 11,261 | 17,280 |
| Automotive07b [confidential] | Automotive | 8,649 | 23,631 | $1.52 \cdot 10^{8}$ |
| Automotive2.4 [47] | Automotive | 18,616 | 350,221 | $1.65 \cdot 10^{1534}$ |
| JavaGC [75] | Compiler | 39 | 105 | 193,536 |
| BerkeleyDBC [75] | Database | 18 | 29 | 2,560 |
| featureide-berkeleydb [64] | Database | 76 | 141 | $4.08 \cdot 10^{9}$ |
| FinancialServices 2018-05-09 [72] | Finance | 771 | 7,241 | $9.74 \cdot 10^{13}$ |
| HiPAcc [75] | Systems Software | 31 | 104 | 13,485 |
| axtls.2.1.4 [75] | Systems Software | 94 | 190 | $1.96 \cdot 10^{12}$ |
| axTLS [47] | Systems Software | 96 | 203 | $8.26 \cdot 10^{11}$ |
| fiasco.17.10 [75] | Systems Software | 234 | 1,178 | $1.02 \cdot 10^{10}$ |
| uClibc-ng.1.0.29 [75] | Systems Software | 269 | 1403 | $8.02 \cdot 10^{36}$ |
| uClibc [47] | Systems Software | 313 | 1,285 | $1.66 \cdot 10^{40}$ |
| toybox.0.7.5 [75] | Systems Software | 316 | 108 | $1.43 \cdot 10^{81}$ |
| uClinux-base [47] | Systems Software | 380 | 7,366 | $2.61 \cdot 10^{022}$ |
| toybox [75] | Systems Software | 544 | 1,020 | $1.44 \cdot 10^{017}$ |
| axTLS [75] | Systems Software | 684 | 2,155 | $4.28 \cdot 10^{020}$ |
| busybox-1.18.0 [47] | Systems Software | 854 | 1,322 | $2.06 \cdot 10^{201}$ |
| busybox.1.28.0 [80] | Systems Software | 998 | 962 | $1.31 \cdot 10^{248}$ |
| am31.sim [47] | Systems Software | 1,178 | 2,845 | $2.62 \cdot 10^{118}$ |
| embtoolkit [47] | Systems Software | 1,179 | 5,967 | $5.13 \cdot 10^{096}$ |
| ref4955 [75] | Systems Software | 1,218 | 3,099 | $8.20 \cdot 10^{123}$ |
| ecos-icse11 [75] | Systems Software | 1,244 | 3,146 | $4.97 \cdot 10^{125}$ |
| pati [75] | Systems Software | 1,248 | 3,266 | $7.90 \cdot 10^{126}$ |
| p2106 [47] | Systems Software | 1,262 | 3,102 | $3.02 \cdot 10^{124}$ |
| integrator.arm9 [75] | Systems Software | 1,267 | 50606 | $4.05 \cdot 10^{129}$ |
| olpce2294 [75] | Systems Software | 1,274 | 3,881 | $8.18 \cdot 10^{126}$ |
| adderII [75] | Systems Software | 1,276 | 3,206 | $3.57 \cdot 10^{125}$ |
| at91sam7sek [75] | Systems Software | 1,296 | 3,921 | $2.58 \cdot 10^{127}$ |
| se77x9 [75] | Systems Software | 1,319 | 49,937 | $1.20 \cdot 10^{135}$ |
| m5272c3 [75] | Systems Software | 1,323 | 3,297 | $2.36 \cdot 10^{125}$ |
| phycore229x [75] | Systems Software | 1,360 | 4,026 | $1.76 \cdot 10^{136}$ |
| freebsd-icse11 [75] | Systems Software | 1,396 | 62,183 | $8.38 \cdot 10^{313}$ |
| ea2468 [47] | Systems Software | 1,408 | 3,470 | $4.81 \cdot 10^{130}$ |
| uClinux-distribution [47] | Systems Software | 1,580 | 2,131 | $4.07 \cdot 10^{409}$ |
| fiasco [75] | Systems Software | 1,638 | 5,228 | $3.58 \cdot 10^{14}$ |
| uClinux [75] | Systems Software | 1,850 | 2,468 | $1.62 \cdot 10^{091}$ |
| linux-2.6.33.3 [47] | Systems Software | 6,467 | 132,032 | unknown |
| busybox-1.18.0 [75] | Systems Software | 6,796 | 17,836 | $8.49 \cdot 10^{216}$ |
| 2.6.28.6-icse11 [75] | Systems Software | 6,888 | 343,944 | unknown |
| uClinux-config [75] | Systems Software | 11,254 | 31,637 | $7.78 \cdot 10^{417}$ |
| buildroot [75] | Systems Software | 14,910 | 45,603 | unknown |
| embtoolkit [75] | Systems Software | 23,516 | 180,511 | $2.10 \cdot 10^{218}$ |
| freetz [75] | Systems Software | 31,012 | 102,705 | unknown |
| 2.6.32-2var [75] | Systems Software | 60,072 | 268,223 | unknown |
| 2.6.33.3-2var [75] | Systems Software | 62,482 | 273,799 | unknown |
| Dune [75] | Solving | 17 | 16 | 2,304 |
| Polly [75] | Team Management | 40 | 100 | 40,000 |
| X264 [75] | Video | 16 | 11 | 1,152 |
| VP9 [75] | Video | 42 | 104 | 216,000 |
| JHipster [75] | Web Application | 45 | 104 | 26,256 |

we consider each d-DNNF compiler appearing in their list, namely c2d [24], dSharp [68], and d4 [56]. For d-DNNF reasoners, no tool capable of reusing d-DNNFs for repetitive queries have been empirically evaluated or considered in another scientific publication, to the best of our knowledge. However, we extended our search to tools without scientific publication and found two tools providing such functionality, namely the reasoner query-ddnnf developed by Caridriot et al.[18] and a new version of d4 on GitHub. The extended (compared to the original work [56]) version of d4, does not allow reasoning on an existing d-DNNF, but allows to compute the number of satisfying assignments for several queries for a given CNF which we also include in our evaluation (referred to as d4-query). We evaluate our proposal ddnnife, query-ddnnf and d4-query as d-DNNF-based reasoners. In preliminary experiments, we found that using query-ddnnf in combination with dSharp always results in cardinalities of zero which may be caused by dSharp violating the standard specified by c2d in some cases. Thus, we exclude the combination of query-ddnnf and dSharp. query-ddnnf does not support the output format of the latest d4 version. Therefore, we evaluate query-ddnnf with an earlier version (d4v1).[19] If not stated otherwise, d4 refers to the latest version (d4v2) during the evaluation.

Table 2. Overview Evaluated Tools

| Solver | Type | Reference | Version/Commit |
|---|---|---|---|
| countAntom | #SAT Solver | [16] | 1.0 |
| Ganak | #SAT Solver | [88] | 18af636[20] |
| sharpSAT | #SAT Solver | [98] | 1.1 |
| c2d | d-DNNF Compiler | [24, 25] | 2.2 |
| d4 | d-DNNF Compiler | [56] | v2 (v1 for query-ddnnf) |
| dSharp | d-DNNF Compiler | [68] | 5a4490[21] |
| ddnnife | d-DNNF Reasoner | This Submission | abfaa7[22] |
| query-ddnnf | d-DNNF Reasoner | No scientific publication | 0.4 |
| d4-query | d-DNNF Reasoner | No scientific publication | v2 |

As baseline for evaluating our approach using d-DNNFs, we include the #SAT solvers sharpSAT, countAntom, and Ganak. In recent work [91], we identified those three solvers as the fastest #SAT for analyzing feature models. Also, Ganak won the model counting competition 2020 [31], while sharpSAT won in 2021 and 2022.[23] Only countAntom and our tool ddnnife support multi-threading. Since other tools could also benefit from parallelization for the queries, we decided to evaluate every tool with a single thread for better comparability. The d-DNNF compilers and #SAT solvers require a CNF as input. We translate every feature model to CNF using FeatureIDE [64] prior to the experiments. This transformation produces an equivalent CNF without introducing any artificial variables [54].

*Experiment 1: Compilation to d-DNNF.* In the first experiment, we measure the runtimes required to translate CNFs representing our subject systems to d-DNNF. Also, we examine the sizes of the resulting d-DNNF. Hereby, we evaluate each d-DNNF compiler, namely c2d, dSharp, and d4, on all 62 subject systems with a timeout of 30 minutes per feature model. The timeout of 30 minutes throughout the experiments is motivated by: (1) we expect that higher runtimes are typically unsuitable for interactive applications and continuous integration pipelines and (2) preliminary experiments showed that further increasing the timeout does not lead to substantial differences

[18] https://www.cril.univ-artois.fr/KC/d-DNNF-reasoner.html
[19] https://www.cril.univ-artois.fr/KC/d4.html
[23] https://mccompetition.org/index.html

in the results. For each combination of compiler and subject system, we perform 50 repetitions. If not stated otherwise, we use the median over the 50 repetitions in plots in this evaluation. The compiled d-DNNFs are used for Experiment 2 and Experiment 3. The insights from Experiment 1 are used to answer RQ1.

*Experiment 2: Cardinality of Features.* In the second experiment, we measure the runtimes required to compute the cardinalities of all features in a given feature model. Hereby, we evaluate each d-DNNF tool combination (consisting of compiler and reasoner) and #SAT solver on each subject system. We set a timeout of 30 minutes for computing the cardinality of all features for a single feature model. For each measurement, we perform 50 repetitions. The insights from Experiment 2 are used to answer RQ2–RQ4. Note that some feature models [75] are only available as propositional formulas in DIMACS format without further information on the semantics of the variables. Hence, we consider the cardinality for each variable here.

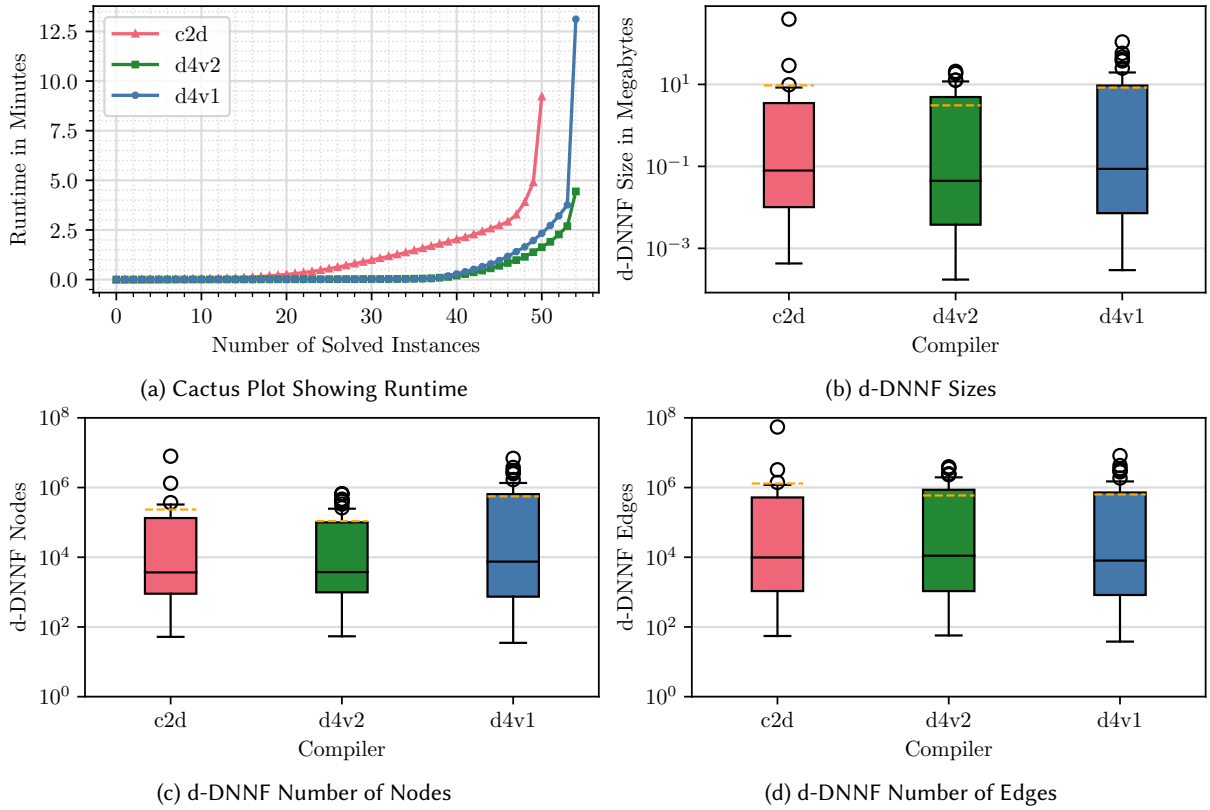
*Experiment 3: Cardinality of Partial Configurations.* In the third experiment, we measure the runtimes required to compute the cardinality of partial configurations for a given feature model. As we have no real queries available, we randomly generate 250 partial configurations for each subject system. We ensure that each partial configuration is satisfiable, as we assume that satisfiable configurations better reflect practical usage of feature-model counting based on the applications found in the literature [93]. To generate the partial configurations, we iteratively select or deselect a randomly chosen feature. If this results in an invalid configuration, we discard the current selection and continue. The 250 configurations are separated in 50 chunks with sizes of 2, 5, 10, 20, and 50 features (randomly included or excluded), respectively. For the ten feature models with fewer than 50 features, we skip chunks exceeding the number of features in the model. With the different configuration sizes, we aim to cover a wide range of analyses dependent on the cardinality of partial configurations [93]. As for Experiment 2, we evaluate each d-DNNF tool combination and #SAT solver on each subject system. We set a timeout of 30 minutes for computing the cardinality of all 250 partial configurations for a single feature model. For each measurement, we perform 50 repetitions. The insights from Experiment 3 are used to answer RQ2–RQ4.

*Statistical Tests.* We use a Wilcoxon signed-ranked test [101] to compare the performance of two tools on the dataset. For comparing multiple tools, we perform a Friedman test followed by a post-hoc Conover test [19]. We use those tests as (1) we do not assume a normal distribution and (2) the runtimes are not independent of each other. Hereby, we always assume a significance level of $\alpha = 0.05$. To examine the effect sizes for significant results, we use Cohen's d [90]. For rating the effect size, we use the classification shown in Table 3 as suggested by Sullivan and Feinn [90]. For instance, we consider a value of $0.5 \leq d < 0.8$ as medium effect size.

| Cohen's $d$ | Effect Size Classification |
|---|---|
| $0.0 \leq d < 0.2$ | Very Small |
| $0.2 \leq d < 0.5$ | Small |
| $0.5 \leq d < 0.8$ | Medium |
| $0.8 \leq d < 1.3$ | Large |
| $1.3 \leq d$ | Very Large |

Table 3. Classification Effect Sizes of Cohen's d Adapted from Sullivan and Feinn [90]

*Technical Setup.* The entire empirical evaluation was performed on an Ubuntu 20.04.3 LTS server with a 64-bit architecture. The processor is an Intel(R) Xeon(R) CPU E5-260v3 with 2.40Ghz clock rate. Overall, 256 GB of RAM were available. During the measurements, the machine was only used for our evaluation to reduce the impact of other processes. For each tool, the memory limit is set to 8 GB as we consider 8 GB to be a reasonable




(a) Cactus Plot Showing Runtime

(b) d-DNNF Sizes

(c) d-DNNF Number of Nodes

(d) d-DNNF Number of Edges

Fig. 7. Results d-DNNF Compilers

limit for hardware (e.g., notebooks or PCs) in practice. Furthermore, in preliminary experiments we found no major differences in measured runtimes if we further increased the memory limit. To measure runtimes, we used a Python-based framework which internally uses the module `timeit`.[24]

## 6.3 Results

*Experiment 1.* Figure 7a shows the runtimes of the d-DNNF compilers for transforming the subject systems as cactus plot. The x-axis indicates the number of solved instances. The y-axis shows the cumulative runtime in seconds. In Experiment 2 & 3 (see below), we observed that `dSharp` produces faulty d-DNNFs and model counts. Hence, we exclude the compilation times of `dSharp` here as well. For 54 out of the 61 subject systems,[25] at least one d-DNNF compiler successfully compiled the CNF into d-DNNF within 30 minutes. `d4` (both versions) successfully compiled 54 feature models while `c2d` successfully compiled 52. For the 52 models successfully evaluated by both compilers, `d4` required 4.62 (14.3 for old version) seconds on average and `c2d` 69.5 seconds.

Figure 7b shows the sizes of the compiled d-DNNFs. For every boxplot, the dashed orange line indicates the average and the solid black line within the box indicates the median. The circles outside the boxes represent

[24] https://docs.python.org/3/library/timeit.html
[25] Failed for: four Linux variants, Embtoolkit, Freetz, and Automotive05.

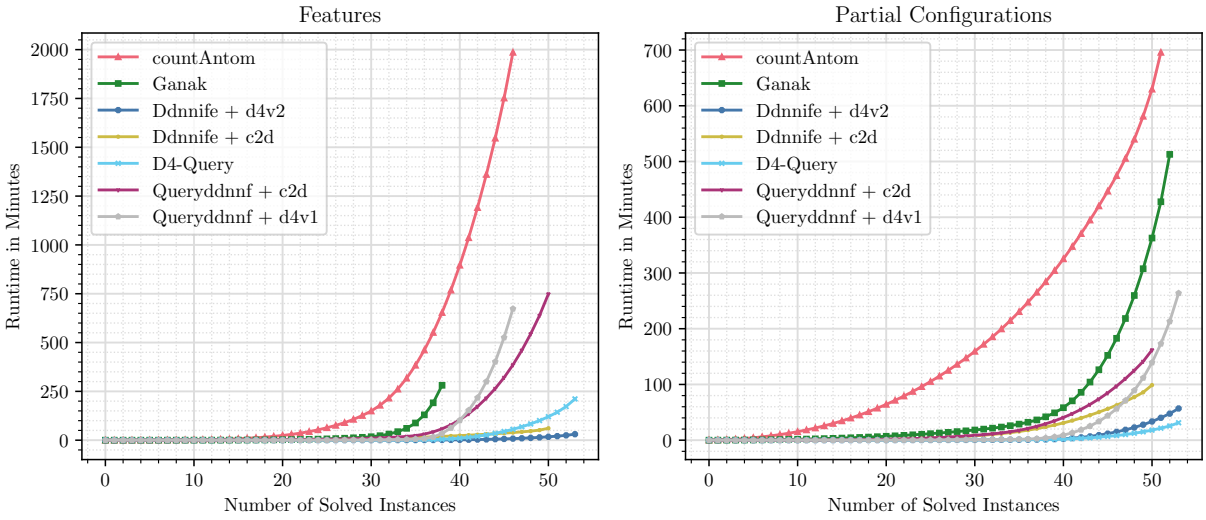



Fig. 8. Cactus Plot Showing Runtime for Offline + Online Phase

outliers. The majority (92.5%) of d-DNNFs require less than 10 MB of space. All but one[26] d-DNNF are smaller than 50 MB. The d-DNNFs produced by the fastest compiler (d4) require at most 20.1 MB (median 0.04 MB). Those d-DNNFs compiled by d4 have 54–665,415 nodes and 57–3.94 · $10^6$ edges. The fact that the number of edges is higher than the number of nodes indicates the reuse of d-DNNF subtrees. For instance, d4 d-DNNFs have 3.4 times as many edges as nodes on average.

*Experiment 2.* All tools except dSharp and sharpSAT computed the same results (i.e., cardinalities) for each feature of all considered feature models. The results of dSharp (as black box) and sharpSAT differed from those results for 57.5% (31 affected models) and 14.8% (8) of the overall computed cardinalities, respectively. ddnnife with d-DNNFs compiled by dSharp differed for 0.54%. Due to the inconsistent results based on dSharp and sharpSAT, we decided to exclude them from runtime comparisons for this and following experiments.

Figure 8 shows a cactus plot indicating the cumulative runtimes required to compute the cardinality of all features and partial configurations using d-DNNFs (offline + online phases) and repetitive #SAT calls. For each feature model successfully compiled to d-DNNF, at least one d-DNNF tool combination was able to successfully compute all feature cardinalities within 30 minutes. The #SAT solvers evaluated all feature cardinalities for only 47 feature models, failing for each feature model the d-DNNF compilers were not able to compile and seven additional ones. For feature models also successfully evaluated by a #SAT solver, applying d-DNNFs is overall 71.4 times faster. If we perform a linear extrapolation from the average time required to evaluate a variable,[27] ddnnife with d4 is up-to ~6,400 (~15,388 without compilation) times faster than the fastest #SAT solver.

Figure 9 shows the relative runtimes of the d-DNNF-based tools compared to the performance of ddnnife. The second row shows only the online phases while the first row also includes compilation times. Note that d4-query has no online phase and the runtimes in the first and second row are equal. Within each box, the dashed line indicates the average and the solid line the median runtime. For ddnnife and query-ddnnf, the runtimes of the respective best performing compiler (d4 and c2d) are included. Overall, ddnnife combined with

[26] Automotive02, compiled by c2d, required 391 MB.

[27] From sample-based tests, we observed that the runtimes are similar for the vast majority of variables. Hence, we consider 153 variables are valid sample for such an extrapolation.

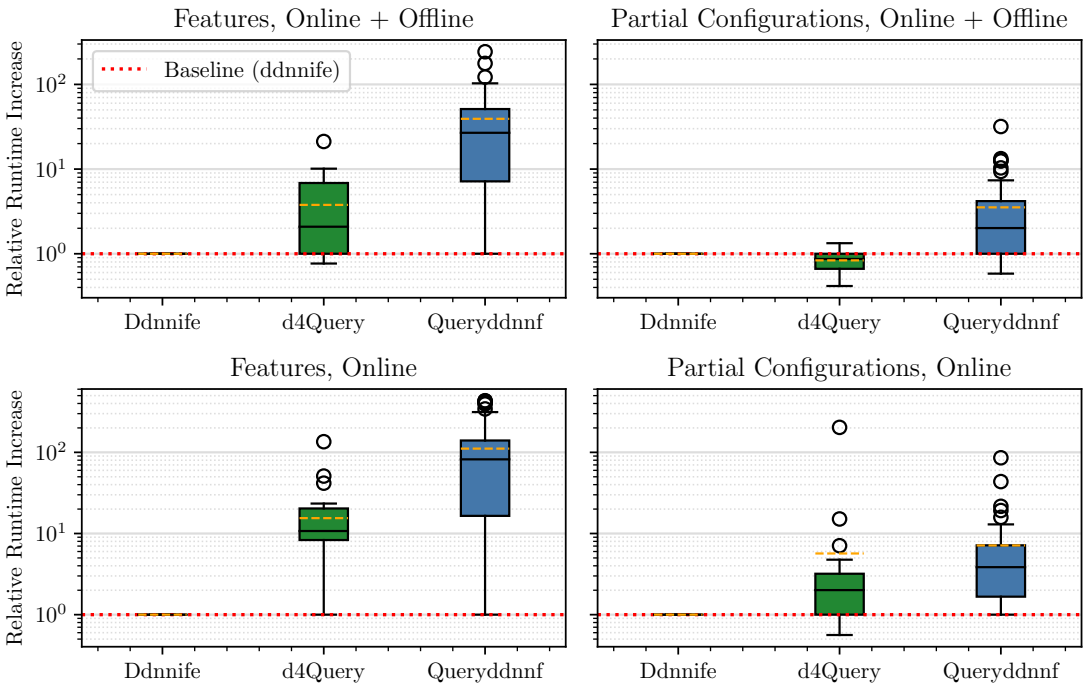



Fig. 9. Runtime Comparison of d-DNNF-based Tools (Runtimes Relative to ddnnife)

d4 is the fastest d-DNNF-based tool combination requiring 6.02 minutes for solving 54 feature models. d4-query requires 40 minutes for the same 54 feature models. This difference between ddnnife and d4-query is significant ($p = 5.3 \cdot 10^{-7}$) with a medium effect size ($d = 0.65$). query-ddnnf with c2d solves all cardinalities for only 53 models and requires 175 minutes for them. For every feature model requiring more than 0.1 seconds to evaluate with all three tool (combinations), ddnnife with d4 requires the least runtime. Excluding compilation time, ddnnife requires 1.31 minutes for all 54 feature models while query-ddnnf requires 72.0 minutes for 53 models.

*Experiment 3.* All tools but dSharp computed the same cardinalities for every partial configuration and feature model. The results of dSharp (black box) and ddnnife with dSharp differed from those results in 41.83% (31 affected models) and 5.83% (31 affected models) cases, respectively. Their runtimes are excluded from the runtime comparisons as in Experiment 2.

The right side of Figure 8 shows the runtimes required to compute the cardinality of the 250 generated partial configurations with d-DNNFs and repetitive #SAT calls. ddnnife and query-ddnnf in combination with any compiler and d4-query are able to compute the cardinality for the 250 partial configurations within 30 minutes for every successfully compiled d-DNNF (overall 54). The fastest #SAT solver Ganak successfully evaluated 53. For those 53 models, the fastest runtimes amount to 3.87 minutes for d-DNNF and 51.8 minutes for #SAT.

The right side of Figure 9 shows the relative runtimes of the d-DNNF-based tools compared to ddnnife for computing the cardinality of partial configurations. For partial configurations, query-ddnnf requires the least overall runtime with 5.88 minutes. ddnnife and query-ddnnf with d4 require 9.24 minutes and 50.7 minutes, respectively. Both, d4-query and ddnnife are significantly faster than query-ddnnf, with $p < 10^{-6}$, $d = 0.80$ and $p < 10^{-4}$, $d = 0.53$, respectively. The difference between ddnnife and d4-query is not significant ($p = 0.33$). Without considering compilation time, ddnnife requires 4.80 minutes and query-ddnnf 37.5 minutes.

## 6.4 Discussion

**RQ1** *How efficient are d-DNNF compilers when applied to industrial feature models regarding runtime and memory?* Our results indicate that compilation to d-DNNF is promising for feature-model counting. Compiling to d-DNNF seems to only fail for feature models that are very hard for feature-model counting in general, as each #SAT failed for the same feature models. This observation matches the expectation as popular d-DNNF compilers and #SAT behave very similarly [24, 56, 68, 98]. In most environments, the memory usage of d-DNNFs should also cause no problems. The fastest compiler `d4` produces d-DNNFs requiring at most 20.1 MB. Overall, considering d-DNNF size and compilation time, `d4` is the most promising d-DNNF compiler for feature models. Nevertheless, since every considered d-DNNF compiler failed for seven of the 61 systems, there is potential and demand for improving the scalability of compilers for feature models. These performance results are in line with previous examinations on the performance of d-DNNF compilers [56, 91, 92]. Hence, our results suggest that the compilers behave similarly to other boolean instances (e.g. formulas representing bayesian networks [56]) when compiling industrial feature models.

**RQ2** *Which tools and tool combinations compute correct cardinalities?* `countAntom`, `Ganak`, `d4`, `c2d`, `ddnnife` (with `c2d` and `d4`), and `query-ddnnf` computed the same cardinalities for every single computation which we consider as a strong indicator for the correctness of these tools. In particular, due to more than 200k consistent results, it is very likely that the algorithms and optimizations presented in this submission and implemented in `ddnnife` yield correct results. Only on faulty d-DNNFs produced by `dSharp`, `ddnnife` computes a few differing cardinalities, which is very likely caused by `dSharp` providing incorrect d-DNNFs in those cases. Due to the incorrect results, we did not consider `sharpSAT` and `dSharp` for runtime comparisons.

As `sharpSAT` has been used for analyzing feature models in various publications [4, 35, 55, 57, 70, 75, 91], our results on its correctness may also have implications for those results. Also, the errors indicate the importance of validating model counting results as recognizing a faulty result is difficult without a reference value.

**RQ3** *How efficient is reusing d-DNNFs for computing cardinalities of features and partial configurations compared to repetitive #SAT calls?* Reusing d-DNNFs for repetitive queries of feature-model counting is substantially faster than using state-of-the-art #SAT solvers for every considered feature model. Often, applying d-DNNFs is multiple orders of magnitude faster than using repetitive #SAT invocations. Overall, we argue the effort of compilation to d-DNNF is worth for feature-model counting and substantially more efficient than the current state-of-the-art of repetitive #SAT queries. As d-DNNFs for repetitive counting queries have not been empirically evaluated in any domain before, our results provide the first insights on the performance. The substantial improvements over repetitively invoking #SAT solvers suggest that reusing d-DNNFs may also be promising for other domains.

**RQ4** *Which d-DNNF-based tools perform best for computing cardinalities?* `ddnnife` with `d4` and `d4-query` appear to be the most promising options for repetitive queries of feature model counting. Overall, the combination of `ddnnife` and `d4` is the fastest requiring 10.8 minutes for computing the cardinalities of features and partial configurations on all feature models followed by `d4-query` (46.9 minutes). For cardinality of features, `ddnnife` significantly outperforms `d4-query`. For the experiments on partial configurations, we have fewer queries and also more assumptions per query. While `ddnnife` is still fastest when a d-DNNF is already given, the performance advantages are smaller. Consequently, more queries are needed to amortize the compilation to d-DNNF when compared to `d4-query`. Thus, when the d-DNNF is not needed for further reuse and only few queries are computed `d4-query` can be advantageous. However, applications suggested in the literature [36, 53, 93] often demand a high number of partial configuration queries [92]. For instance, supporting the interactive configuration process by providing an indication on the impact of decisions on the configuration space may require up-to $O(|features|^2)$ queries [62, 92]. We argue that `ddnnife` is likely to amortize the compilation cost compared to `d4-query` for such applications. Still, it may be beneficial to further improve the performance of `ddnnife` for partial configurations. One option may be sorting queries to reduce the share of nodes adapted by subsequent queries. Furthermore,

partial derivatives could be also applied for partial configurations if only one literal is flipped from a previous partial configuration.

## 6.5 Threats to Validity

*Translating Feature Models to CNF.* Each evaluated compiler and #SAT solver uses CNF as input format. To employ the tools, the feature models need to be translated to CNF. Two equivalent but syntactically different CNFs may result in significantly varying runtime for solvers and compilers [46, 54, 75]. Hence, changing the CNF translation may have an impact on the measured runtimes. However, analyzing multiple CNF translations is beyond the scope of this work. Another aspect to consider is that some CNF translations are equisatisfiable but do not preserve the number of satisfying assignments. Such a translation may lead to faulty cardinalities in our analyses. For each feature model given in a proprietary format, we use the transformation to CNF employed by FeatureIDE [64] which preserves equivalence and, thus, the number of satisfying assignments [54]. The remaining feature models (collected by Oh et al. [76]) were already specified in CNF and have been used in other empirical evaluations [44, 75, 76, 82].

*Parameterization of Solvers and Compilers.* All evaluated #SAT solvers and compilers provide parameters that potentially impact their runtime. It is possible that different parameters may significantly change the performance of evaluated tools. However, we expect that using default parameters reasonably represents the usage in practice. Parameter optimization for each solver and compiler is out of scope for this work.

*Random Effects and Computational Bias.* When measuring the runtime of a solver required to analyze a feature model, the runtimes may vary between measurements due to computational bias, random effects, or background processes. In particular, the performance of `c2d` is reliant on an initial decomposition tree which is heavily dependent on random decisions. To reduce the impact of such influence factors, we perform 50 repetitions for each measurement present the median if not stated otherwise. Also, while performing measurements no other major processes were executed. Furthermore, we employ established statistical tests to validate the significance of our conclusions.

*Memory Limit.* For each tool, we limited the memory usage to 8 GB due to expected requirements in practice and preliminary experiments (cf. Section 6.2 for more details). Still, changing the memory limit may have an impact on the performance of the different tools. However, repeating all experiments with multiple memory limits would have required vast computational resources and further increase the complexity of the evaluation.

*External Validity - Selection of Queries.* For evaluating the tools on computing cardinality of features, we consider the cardinality of all features. Considering fewer features may have an impact on the performance of the different tools. However, many analyses considered in the literature often require to analyze many or even all features [93]. In particular, the metric *homogeneity*, which has often been considered [18, 29, 30, 37, 93], by definition requires to compute the cardinality of all features. Also, `ddnnife` amortizes the initial cost of compiling after few queries compared to #SAT solvers.

For cardinality of partial configurations, we use randomly generated configurations of different sizes. It is possible that the random decisions do not fully reflect the queries required for specific analyses in practice. However, the different analyses dependent on partial configurations require multiple settings with varying number of (de-)selected features [53, 75, 93]. With our evaluation, we aim to not be limited to single analyses. Due to the heterogeneity of configurations in different analyses and real partial configurations typically not being available, we consider random configurations of various sizes as reasonable decision for our experiments.

*External Validity - Tools.* The measured performance of the evaluated tools cannot be necessarily transferred to other tools. However, to the best of our knowledge, we evaluated all publicly available d-DNNF compilers

and reasoners. For #SAT solvers, we selected the two best performing #SAT solvers for feature-model counting according to our previous experiments [94] and the winning #SAT solver of the model counting competition 2020. Hence, we expect our results to reasonably represent the performance of both approaches.

*External Validity - Subject Systems.* Our conclusions do not necessarily hold for other real-world feature models. However, we evaluated feature models from a variety of domains, namely systems software, automotive, financial services, and multiple other domains. Furthermore, we covered a wide range of features (11–62,482), clauses (1–350,221), number of valid configurations (1,024–$10^{1534}$), and induced runtimes for solving #SAT (between few milliseconds and hitting timeouts of 24 hours [94]).

## 7 RELATED WORK

In this section, we discuss work that (a) performs model counting with d-DNNFs, (b) uses d-DNNFs in the context of feature models, (c) proposes alternative approaches to allow feature-model counting, (d) applies knowledge compilation in feature-model analysis, or (e) presents other knowledge-compilation artifacts for which model counting is tractable.

*Model Counting with d-DNNFs.* Each considered d-DNNF compiler, namely `c2d` [24], `dSharp` [68], and `d4` [56], is able to compute the number of satisfying assignments of the input. However, neither compiler allows reusing a given d-DNNF to compute cardinalities of features or partial configurations. While `d4` (given a CNF) and `query-ddnnf` support computing multiple queries, those functionalities have not been considered in a scientific publication and have not been evaluated before. Generally, there has not been any empirical evaluation on reusing d-DNNFs for counting.

*d-DNNF Exploitation in Feature-Model Analysis.* Sharma et al. [87] use d-DNNFs for uniform random sampling with their tool *KUS*. Internally, *KUS* depends on counting on a given d-DNNF. While that counting procedure could be extracted, it does not support applying assumptions (here: including/excluding features) and, thus, cannot be used for both query types we consider. Bourhis et al. [15] employ d-DNNFs for multiple different feature-model analyses including counting. However, for counting, they only consider single computations for the entire feature model while we target repetitive queries for features and partial configurations. In previous publications, we [94] and Kübler et al. [53] evaluate d-DNNF compilers for the task of computing the number of valid configurations. However, in both publications the compilers are just used as black-box #SAT solvers, essentially compiling a new d-DNNF for every single computation, and do not reuse the compiled d-DNNF. Furthermore, no repeated queries (e.g., for cardinality of features) are performed.

*Solutions for Feature-Model Counting.* In the above mentioned works, Kübler et al. [53] and we [94] compute the cardinalities of industrial feature models. Both works are limited to one query per feature model and only use the solvers as black-box, whereas we reuse the compiled the d-DNNFs to reduce the effort of repetitive queries.

Pohl et al. [84] compare the runtimes of several solver types, namely BDD, SAT, and CSP, on different feature-model analyses, including feature-model counting. The results indicate that BDDs scale substantially better than SAT and CSP for computing the number of valid configurations. However, their dataset contains only comparatively small feature models with less than 300 features, whereas our benchmark contains feature models with several ten-thousands features. In previous work, we found that compiling to BDD scales for only very few industrial feature models and considerably worse than d-DNNFs [39, 91].

Other work proposes ad-hoc solutions for feature model counting targeting feature models with no [38] or very few cross-tree constraints [30]. While those perform well for the feature models with no or very few cross-tree constraints, recent work indicates that industrial feature models tend to have very large number of cross-tree constraints (e.g., up-to 15k in our dataset) [39, 48, 75, 94].

*Uniform-Random Sampling.* Several approaches for uniform random sampling internally depend on counting [4, 70, 76, 87]. For instance, *Smarch* [75] builds a uniform sample by repetitively invoking `sharpSAT`. *Spur* [4] adapts `sharpSAT` to cache values of computation nodes to accelerate deriving valid configurations. However, like for *KUS* (as we described in *d-DNNF Exploitation in Feature-Model Analysis*), it is not possible to apply assumptions for *Spur*. Hence, computing the cardinality of features or partial configurations is also not possible without major conceptual adaptations.

*Knowledge Compilation in Feature-Model Analysis.* Binary decision diagrams (BDDs) are often applied for feature-model analyses [1, 3, 10, 67, 74, 94], including some proposals to use BDDs for feature-model counting [39, 42, 65, 84, 94]. However, BDDs fail to scale for many industrial feature models [39, 75, 94, 95]. Krieter et al. [51, 52] employ modal implication graphs to accelerate decision propagation during product configuration. However, modal implication graphs are not tractable for feature-model counting.

*Model Counting with Other Knowledge-Compilation Artifacts.* Model counting is tractable for several knowledge-compilation artifacts (i.e., has a polynomial time complexity with respect to the size of artifacts). According to the knowledge compilation map of Darwiche and Marquis [26], binary decision diagrams, d-DNNFs, and MODs (a DNF that satisfies smoothness and determinism) allow polynomial-time counting. In other work, sentential decision diagrams [78] and affine decision trees [49] have been found tractable for model counting. While all these formats allow counting in theory, we compared the performance of BDD, SDD, EADT, and d-DNNF compilers for the task of feature-model counting in recent work [91] and found d-DNNF compilers to be the most efficient.

*Incremental Solving.* An incremental solver performs multiple queries on highly similar formulas while keeping information between those queries to accelerate later invocations [28]. For regular SAT, several solvers were proposed with promising results for many use-cases [27, 28, 71]. Typically, those solvers are adaptations of popular SAT solvers based on DPLL [33] or CDCL [61]. While such incremental SAT solvers have been used for years, there have not been such adaptations for #SAT.

## 8 CONCLUSION

Many analyses based on feature-model counting require various similar counting queries to compute the number of valid configurations that contain a certain feature or conform to a partial configuration. We are the first to realize reusing d-DNNFs to reduce the effort of repetitive counting queries. Our evaluation shows that reusing d-DNNFs substantially reduces the runtime compared to the state-of-the-art of repetitively invoking #SAT solvers. We found that compilation to d-DNNF scales for every feature model for which state-of-the-art #SAT solvers are able to compute a result. Applying d-DNNFs is faster than #SAT for every considered feature model, that at least one tool was able to analyze, reducing runtimes by more than 99.9% in the best case including compilation time. Further, our tool `ddnnife` combined with the compiler `d4` was the overall fastest tools for the considered problems, namely cardinality of features and cardinality of partial configurations. We therefore recommend reusing d-DNNFs with `ddnnife` when many queries of feature-model counting are required. When only few queries are required and the d-DNNF is not further reused, using `d4-query` can yield benefits over `ddnnife`.

## REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)* 78, 6 (2013), 657–681.

[3] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature Model Differences. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*. Springer, 629–645.

[4] Dimitris Achlioptas, Zayd S Hammoudeh, and Panos Theodoropoulos. 2018. Fast Sampling of Perfectly Uniform Satisfying Assignments. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 135–147.

[5] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. 2009. On the Structure of Industrial SAT Instances. In *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Springer, 127–141.

[6] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal (SQJ)* 19, 3 (2011), 579–612.

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

[8] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. 2000. Counting Models Using Connected Components. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 157–162.

[9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[10] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Technical Report 2007-01, Lero, 129–134.

[11] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*. 677–682.

[12] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 302–319.

[13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.

[14] Armin Biere. 2008. PicoSAT Essentials. *J. Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

[15] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. 2023. *Reasoning on Feature Models: Compilation-Based vs. Direct Approaches.* Technical Report. Cornell University Library.

[16] Jan Burchard, Tobias Schubert, and Bernd Becker. 2015. Laissez-Faire Caching for Parallel #SAT Solving. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 46–61.

[17] Sheng Chen and Martin Erwig. 2011. Optimizing the Product Derivation Process. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 35–44.

[18] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report. Carnegie Mellon University.

[19] William J Conover and Ronald L Iman. 1981. Rank Transformations as a Bridge Between Parametric and Nonparametric Statistics. *The American Statistician* 35, 3 (1981), 124–129.

[20] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proc. Int'l Workshop on Software Factories (SF)*. 16–20.

[21] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 23–34.

[22] Adnan Darwiche. 2001. Decomposable Negation Normal Form. *J. ACM* 48, 4 (2001), 608–647.

[23] Adnan Darwiche. 2001. On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision. *J. Applied Non-Classical Logics* 11, 1-2 (2001), 11–34.

[24] Adnan Darwiche. 2002. A Compiler for Deterministic, Decomposable Negation Normal Form. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press, 627–634.

[25] Adnan Darwiche. 2004. New Advances in Compiling CNF to Decomposable Negation Normal Form. In *Proc. Europ. Conf. on Artificial Intelligence*. IOS Press, 318–322.

[26] Adnan Darwiche and Pierre Marquis. 2002. A Knowledge Compilation Map. *J. Artificial Intelligence Research (JAIR)* 17, 1 (2002), 229–264.

[27] Niklas Eén and Niklas Sörensson. 2003. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science (ENTCS)* 89, 4 (2003), 543–560.

[28] Katalin Fazekas, Armin Biere, and Christoph Scholl. 2019. Incremental Inprocessing in SAT Solving. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, Mikoláš Janota and Inês Lynce (Eds.). Springer, 136–154.

[29] David Fernández-Amorós, Ruben Heradio Gil, and José Antonio Cerrada Somolinos. 2009. Inferring Information From Feature Diagrams to Product Line Economic Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 41–50.

[30] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. 2014. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *IEEE Trans. on Software Engineering (TSE)* 40, 9 (2014), 895–910.

[31] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. 2021. The Model Counting Competition 2020. *ACM J. of Experimental Algorithmics (JEA)* 26, Article 13 (2021), 26 pages.

[32] Claudia Fritsch, Richard Abt, and Burkhardt Renz. 2020. The Benefits of a Feature Model in Banking. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 9, 11 pages.

[33] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL (T): Fast Decision Procedures. In *Proc. Int'l Conf. on Computer Aided Verification (CAV)*. Springer, 175–188.

[34] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2022. Generic Solution-Space Sampling for Multi-Domain Product Lines. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM.

[35] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the Variability Mismatch Between Problem and Solution Space. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 322–333.

[36] Ruben Heradio, David Fernández-Amorós, José Antonio Cerrada, and Ismael Abad. 2013. 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)* 23, 08 (2013), 1177–1204.

[37] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the Statistical Analysis of Variability Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 843–853.

[38] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting Commonality-Based Analysis of Software Product Lines. *IET Software* 5, 6 (2011), 496–509.

[39] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 131–135.

[40] Jinbo Huang and Adnan Darwiche. 2005. DPLL With a Trace: From SAT to Knowledge Compilation. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, Vol. 5. Professional Book Center, 156–162.

[41] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *J. Systems and Software (JSS)* 83, 3 (2010), 485–501.

[42] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. 2012. A Technique for Agile and Automatic Interaction Testing for Product Lines. In *Proc. Int'l Conf. on Testing Software and Systems (ICTSS)*. Springer, 39–54.

[43] David S Johnson. 1992. The NP-Completeness Column: An Ongoing Guide. *J. Algorithms* 13, 3 (1992), 502–524.

[44] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1084–1094.

[45] Vladimir Klebanov, Norbert Manthey, and Christian Muise. 2013. SAT-Based Analysis and Quantification of Information Flow in Programs. In *Proc. Int'l Conf. on Quantitative Evaluation of Systems (QEST)*. Springer, 177–192.

[46] Will Klieber and Gihwon Kwon. 2007. Efficient CNF Encoding for Selecting 1 From n Objects. In *Proc. Int'l Workshop on Constraints in Formal Verification (CFV)*. Springer.

[47] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. 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, 291–302.

[48] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2018. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Software Engineering (SE)*, Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer (Eds.). Gesellschaft für Informatik, 53–54.

[49] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. 2013. Knowledge Compilation for Model Counting: Affine Decision Trees. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press.

[50] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 132–143.

[51] Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. 2021. Incremental Construction of Modal Implication Graphs for Evolving Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 64–74.

[52] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions With Modal Implication Graphs. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 898–909.

[53] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 44–53.

[54] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 110:1–110:13.

[55] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2023. Tseitin or Not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Software Engineering (SE)*, Vol. P-322. Gesellschaft für Informatik, 83–84.

[56] Jean-Marie Lagniez and Pierre Marquis. 2017. An Improved Decision-DNNF Compiler. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. International Joint Conferences on Artificial Intelligence, 667–673.

[57] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 91–100.

[58] Michael Lienhardt, Ferruccio Damiani, Einer Broch Johnsen, and Jacopo Mauro. 2020. Lazy Product Discovery in Huge Configuration Spaces. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 1509–1521.

[59] Jackson A. Prado Lima, Willian D. F. Mendonça, Silvia R. Vergilio, and Wesley K. G. Assunção. 2020. Learning-Based Prioritization of Test Cases in Continuous Integration of Highly-Configurable Software. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 31, 11 pages.

[60] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 136–150.

[61] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. IOS Press, 131–153.

[62] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. 2014. Recommendation Heuristics for Improving Product Line Configuration Processes. In *Recommendation Systems in Software Engineering*. Springer, 511–537.

[63] John McGregor. 2010. Testing a Software Product Line. In *Testing Techniques in Software Engineering*. Springer, 104–140.

[64] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer.

[65] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 761–762.

[66] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 231–240.

[67] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. 2008. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 13–22.

[68] Christian Muise, Sheila McIlraith, J Christopher Beck, and Eric Hsu. 2010. Fast d-DNNF Compilation With sharpSAT. In *Proc. Conf. on Artificial Intelligence (AAAI)*. AAAI Press.

[69] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Advances in Artificial Intelligence*, Leila Kosseim and Diana Inkpen (Eds.). Springer, 356–361.

[70] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 289–301.

[71] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Ultimately Incremental SAT. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 206–218.

[72] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 188–201.

[73] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2022. Guiding the Evolution of Product-Line Configurations. *Software and Systems Modeling (SoSyM)* 21 (2022), 225–247. Issue 1.

[74] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2016. *Finding Product Line Configurations With High Performance by Random Sampling*. Technical Report. University of Texas at Austin, Department of Computer Science.

[75] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.

[76] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Margaret Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. University of Texas at Austin, Department of Computer Science.

[77] Umut Oztok and Adnan Darwiche. 2014. On Compiling CNF Into Decision-DNNF. In *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*. Springer, 42–57.

[78] Umut Oztok and Adnan Darwiche. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 3141–3148.

[79] Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 18, 9 pages.

[80] Tobias Pett, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. AutoSMP: An Evaluation Platform for Sampling Algorithms. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 41–44.

[81] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 78–83.

[82] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. 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, 240–251.

[83] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[84] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. 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, 313–322.
[85] Steven David Prestwich. 2009. CNF Encodings. *Handbook of Satisfiability* 185 (2009), 75–97.
[86] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 20–28.
[87] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S Meel. 2018. Knowledge Compilation Meets Uniform Sampling. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. EasyChair, 620–636.
[88] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, Vol. 19. AAAI Press, 1169–1176.
[89] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 6, 9 pages.
[90] Gail M Sullivan and Richard Feinn. 2012. Using Effect Size—or Why the P Value Is Not Enough. *Journal of Graduate Medical Education* 4, 3 (2012), 279–282.
[91] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28, 29 (2023), 38.
[92] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2023. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *Annals of Mathematics and Artificial Intelligence (AMAI)* (2023).
[93] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 12, 10 pages.
[94] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 3, 9 pages.
[95] Thomas Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 16, 6 pages.
[96] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45.
[97] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 254–264.
[98] Marc Thurley. 2006. sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 424–429.
[99] Leslie G Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. on Computing* 8, 3 (1979), 410–421.
[100] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.
[101] R. F. Woolson. 2008. *Wilcoxon Signed-Rank Test*. John Wiley & Sons. 1–3 pages.
[102] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In *Proc. Int'l Conf. on Formal Engineering Methods (ICFEM)*. Springer, 115–130.