Efficient Slicing of Feature Models via Projected d-DNNF Compilation

Chico Sundermann
University of Ulm
Ulm, Germany

Jacob Loth
University of Ulm
Ulm, Germany

Thomas Thüm
Paderborn University
Paderborn, Germany

ABSTRACT

Configurable systems often contain components from different fields or disciplines that are relevant for distinct stakeholders. For instance, tests or analyses targeting interactions of the software of a cyber-physical system may be only applicable for software components. However, managing such components in isolation is not trivial due, for instance, interdependencies between features. Feature models are a common formalism to specify such dependencies. Feature-model slicing corresponds to creating a subset of the feature model (e.g., with only components relevant to a particular stakeholder) that still preserves transitive dependencies from discarded features. However, slicing is computationally expensive and subsequent analyses often depend on complex computations, such as SAT or #SAT. With knowledge compilation, the original feature model can be translated to a beneficial format (e.g., d-DNNF or BDD) with an initial effort that accelerates subsequent analyses. Consequentially, acquiring a sliced target format depends on two expensive subsequent algorithms. In this work, we merge both steps by proposing *projected d-DNNF compilation*; a novel way to slice feature models that coincidently performs knowledge compilation to d-DNNF. Our empirical evaluation on real-world feature models shows that our tool `pd4` often reduces runtimes substantially compared to existing techniques and scales to more input instances.

KEYWORDS

feature models, configurable systems, product lines, d-DNNF, knowledge compilation, slicing, projection

ACM Reference Format:
Chico Sundermann, Jacob Loth, and Thomas Thüm. 2024. Efficient Slicing of Feature Models via Projected d-DNNF Compilation. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695594

1 INTRODUCTION

Developing configurable systems demands various tests or analyses during different stages of a system's lifecycle [1, 50, 71]. Typically, not every configuration of features (i.e., decomposable units of the system) represents a desired product because of constraints such as dependencies between those features [7]. Hence, performing tests or analyses reliably requires configuration knowledge [7], separating valid from invalid configurations. Feature models are an expressive and commonly used [29, 47] formalism, which serves as a formal specification and an overview for stakeholders.

In many cases, one is interested in analyzing only a subset of a configurable system [2, 28, 29, 62–64, 75]. For instance, in a cyber-physical system, some analyses may only be reasonable for software-related features. Still, in practice there are often constraints over non-software features that transitively influence the valid configurations of the software. Neglecting those constraints may produce invalid configurations considering the original feature model and, consequentially, to inconsistent analysis results. Feature-model slicing corresponds to deriving a feature model with only features of interest but keeping transitive constraints [2, 39, 63]. For the example, slicing can be used to create a feature model with only software-related features that accepts the same software configurations as the overall feature model.

Current techniques for feature-model slicing face scalability issues for larger feature models [2, 39]. In particular, the currently most scalable solution seems to be slicing by applying logical resolution to remove features [39]. Here, features are removed one-by-one which is computationally expensive when removing larger parts of the feature model. However, common use cases for slicing, such as providing views [62], or feature-model interfaces [63], or representing solution space [29], often require slicing large parts of the feature model, which limits the scalability of existing approaches.

Many analyses on the configuration knowledge depend on, potentially numerous, complex computations [71], such as SAT [7, 38, 65] and #SAT [31, 41, 72]. For instance, homogeneity, a metric that indicates the similarity of configurations, requires a #SAT computation for every single feature [30, 71]. When facing multiple computations, knowledge compilation can be applied [20, 71]. Here, the feature model is translated to another format with beneficial properties (e.g., BDDs [13]) in an offline phase. Then, different analyses can be more efficiently performed on the target format in an online phase [20]. A format that gained popularity for analyzing configurable systems over the last few years is the determnistic decomposable negation normal form (d-DNNF) [12, 17, 41, 66, 73]. The format supports linear-time (w.r.t. d-DNNF size) computations for analyses based on SAT, #SAT, and generating configurations [20, 71]. Recent results show that d-DNNFs can vastly improve the scalability of such analyses on configurable systems [12, 66, 73].

However, compiling to d-DNNF is typically expensive and does not scale for some complex systems, such as the Linux kernel [71]. Since slicing and d-DNNF compilation both fail to scale to more complex systems in practice [39, 71], the applicability of analyses relying on these techniques is still limited.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695594

In this work, we propose to combine the concept of slicing and knowledge compilation by introducing *projected d-DNNF compilation*. The idea is to accelerate the slicing process and directly produce a more beneficial representation (i.e., a d-DNNF). In particular, the resulting d-DNNF enables linear-time (w.r.t. number of nodes in the d-DNNF) analyses based on counting [73], sampling [66], or validity [12] on the feature-model slice. We implemented a prototype by adapting the regular d-DNNF compiler `d4` [45] which appears to outperform other compilers on feature models [71].

We study the performance of projected d-DNNF compilation compared to the state of the art in a large-scale empirical evaluation. As subject systems, we consider real feature models and slices applied in practice from our industry partner, feature models from literature [37, 55], and instances from the Model Counting Competition [25]. We compare our approach to slicing with resolution [39] and projected model counters [26]. The latter base on similar techniques as projected d-DNNF compilation, but produce only the projected model count instead of a reusable d-DNNF.

Overall, we make the following three contributions:

- **Projected d-DNNF Compilation.** We propose and present the concept of projected d-DNNF compilation which adapts regular d-DNNF compilation by slicing during the process.
- **Optimizations.** We present various heuristics that we apply to increase efficiency of the compilation, as the scalability of reasoning engines, such as knowledge compilers, heavily depends on employed heuristics [44, 45, 69].
- **Prototype.** We implement the concept and the heuristics by adapting the state-of-the-art compiler `d4`.[1]
- **Empirical Evaluation.** We compare the performance of our projected d-DNNF compiler `pd4` to the state of the art in feature-model slicing and projected model counting.[2]

2 MOTIVATING EXAMPLE

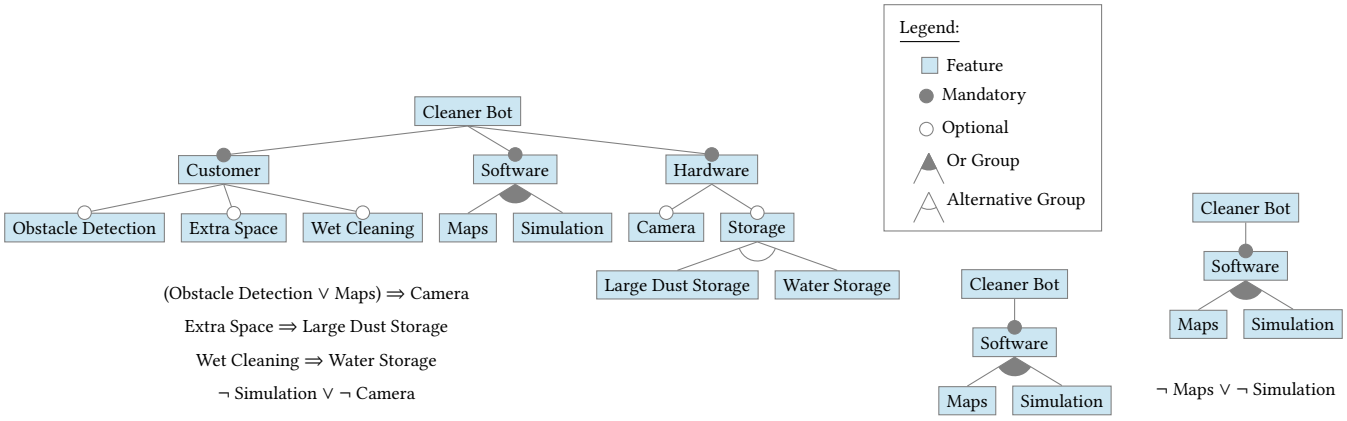
In the following, we provide a short example showcasing the benefits of slicing [2, 39]. Figure 1a depicts an exemplary feature model for a cleaning robot system. The configurable system consists of three major parts: `Customer`, `Software`, and `Hardware`. Each part must be configured as seen by their *mandatory* flag. A customer can select any combination of the three options `Obstacle Detection`, `Extra Space`, and `Wet Cleaning` as denoted by the *optional* flags. The *or*-relation below `Software` denotes that `Maps`, or `Simulation` or both can be selected. Further, a cleaner robot may have a `Camera` and exactly one `Storage` type as denoted by the *alternative*-relation. There are also several cross-tree constraints. For instance, `Wet Cleaning` requires having a `Water Storage`.

Typically, different stakeholders care only about specific parts of the configurable system. For instance, a software engineer may want to explicitly test variability only in source code without having to manage hardware or customer requirements. A typical use case for tests or analyses limited only parts of a system are software-in-the-loop tests [21], which examine software in a simulated environment without the designated hardware components. However, discarding the remaining features is not trivial. With the naive approach of just removing unwanted features and dependencies between them, transitive constraints can be overlooked. In our case, `Maps` and `Simulation` would be simultaneously selectable with the naive approach as seen in Figure 1b. In contrast, this combination would not be valid for the entire feature model due to interaction between `Camera`, `Simulation`, and `Maps` from the cross-tree constraints. ¬ `Simulation` ∨ ¬ `Camera` implies that only one of those can be selected, but `Maps` needs a `Camera` as indicated by the first cross-tree constraint. Consequentially, there are configurations that contain both `Maps` and `Simulation` that can be derived from this naive feature model, but are invalid for the complete feature model. This mismatch may induce problems when using or analyzing the naive model. For instance, the developer may build a test case for a software variant including both `Simulation` and `Camera`, which does not appear in any product of the system.

Slicing a feature model produces a sub feature model with transitive constraints as shown Figure 1c. Here, the effect of the interdependencies on the remaining features is specified as new constraint ¬ Maps ∨ ¬ Simulation. Similar problems also occur for the customer features. Only considering the `Customer` subtree, any combination of the three features can be selected. However, in practice, `Extra Space` and `Wet Cleaning` cannot be selected simultaneously due to storage types being alternative. While the described transitive effects could be recognized by humans for our small example, feature models in practice often have thousands of features and cross-tree constraints [8, 37, 41, 47]. Hence, automated tool support is needed to compute slices and also for subsequent analyses.

Various automated analyses have been proposed for feature models. As the output of slicing is also a feature model, these analyses are also applicable on the slice. Many of those analyses rely on solving complex computational problems [71], such as validity (SAT), counting (#SAT), and enumerating configuration (AllSAT). For instance, customers are likely only provided the `Customer` features for configuring their `Cleaner Bot`. Hence, using a model sliced on the customer features would be sensible for guiding the configuration process with decision propagation [40] or recommendations [49, 57]. A potential use case for counting on a sliced feature model is prioritizing features that appear in many products from the customers point of view [72]. Also, when testing software, it is common to create a sample of configurations [23, 38, 54] from a sliced software feature model [28], as the other features are not relevant for the software tests. The three listed computational problems, namely SAT, #SAT, and creating a sample of configurations, can be solved in linear time on a d-DNNF with respect to its number of nodes [20]. Furthermore, many automated analyses on configurable systems from various areas of software engineering rely on numerous of such complex computations [71]. For instance, providing information on selectable features during the configuration process with decision propagation requires $O(|\text{features}|)$ SAT invocations *per configuration step* [40]. Homogeneity [30, 72], a metric indicating the similarity of configurations, requires a #SAT invocation for every feature. Hence, having a projected d-DNNF can be beneficial to compute a feature-model slice and perform subsequent analyses on it.

[1] https://github.com/SoftVarE-Group/pd4
[2] https://doi.org/10.5281/zenodo.13752743



(a) Original **(b) Naive Submodel** **(c) Sliced Submodel**

Figure 1: Slicing a Cleaner Bot Product Line to Its Software Components

3 BACKGROUND

In this section, we introduce the following concepts that are relevant for this work: feature models, slicing, and d-DNNFs.

3.1 Feature Models

A *configurable system* is a family of related products that share certain *features*. Typically, not every *configuration* of features results in a desired product of the system. In this work, we use *feature models* as formalism to specify the set of valid configurations over those features [7, 36, 51]. The most common notation are *feature diagrams* which define features and dependencies between them using a tree hierarchy combined with cross-tree constraints. Our motivating example Figure 1a is visualized as feature diagram.

Formally, we use a much more generic specification of feature models without a concrete hierarchy. Rather, we define feature models $FM = (N, F)$ as tuple where N is the set of features and F a Boolean formula describing the set of valid configurations. Hence, our findings are applicable also other representations of configuration knowledge as long as they can be reduced to Boolean constraints over the features. We limit the expressiveness to Boolean logic here, since cross-tree constraints are commonly restricted to Boolean logic in the literature [5, 42, 51] and translations from the tree hierarchy of feature diagrams to Boolean logic are well-known [16, 51]. Furthermore, there have been efforts to translate higher-order logics to Boolean logic [53]. During this work, we consider each formula to be in *conjunctive normal form* (CNF) for simplicity. A CNF is a conjunction of clauses where each clause is a disjunction of literals. CNFs are commonly used as input for reasoning engines and translations from feature models to CNF are well-known [42]. A *configuration* $C = (I, E)$ with $I, E \subseteq N$ and $I \cap E = \emptyset$ is *valid* if and only if the formula $F \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e$ is satisfiable. With $\mathbb{V}_{FM}$, we denote the set of valid configurations described by FM.

3.2 Slicing

Feature-model slicing refers to removing features from a feature model while preserving transitive dependencies [2, 39]. In the logic domain, the same concept is also known as *projection* [26] and *forgetting* [20]. We consider every removed variable $n \in N \setminus N'$ as *sliced* and every kept variable $n \in N'$ as *projected* during the remainder of this work. Formally, we call $FM' = (N', F')$ a slice of a feature model $FM = (N, F)$ if $N' \subset N$, Equation 1, and Equation 2 hold. First, if we reduce a valid configuration for FM by removing features that are not part of N', it is valid for the slice FM'. Second, for every valid configuration of FM' we can find at least one valid configuration for FM by selecting or deselecting sliced features.

$$\forall (I, E) \in \mathbb{V}_{FM} : (I \cap N', E \cap N') \in \mathbb{V}_{FM'} \tag{1}$$

$$\forall (I', E') \in \mathbb{V}_{FM'}, \exists (I, E) \in \mathbb{V}_{FM} : I' \subseteq I, E' \subseteq E \tag{2}$$

3.3 Current Approaches for Slicing

In this section, we very shortly present existing approaches for feature-model slicing. For a more thorough comparison, we refer to the work targeting those approaches [2, 39, 75].

Existential Quantification. With existential quantification, a variable is sliced by duplicating the original formula and replacing the variable with true and false in the formulas, respectively [39, 75]. The approach bases on the equivalence $\phi \equiv (\phi \wedge v) \vee (\phi \wedge \bar{v})$. In the left subformula, every v can then be replaced with $\top$ and $\bar{v}$ with $\bot$ and vice versa for the right subformula, resulting in a sliced formula $\phi_v \vee \phi_{\bar{v}}$ without the variable v. Then, the inserted $\top$, and $\bot$ can be resolved with unit propagation. The procedure can be optimized by only duplicating clauses that contain the current variable v. Still, for a slice $FM' = (N', F')$ of $FM = (N, F)$, we need to apply existential quantification for every single sliced variable $v \in N \setminus N'$, which increases the required effort for every variable.

Logical Resolution. With *logical resolution*, we can slice variables v by *resolving* each pair of clauses where one contains a positive literal v and the other contains a negative literal $\neg v$ [39]. Equation 3 shows two clauses with v and $\neg v$ respectively which equivalently resolve to the clause below which does not include v. In the formula, we then remove the premise clauses and add the resolvent clauses.

We need to resolve $|C_v| * |C_{\bar{v}}|$ clause pairs for each variable to slice, where C_v are clauses with v and $C_{\bar{v}}$ clauses with $\bar{v}$. Hence, just as for existential quantification, the required effort increases with the number of sliced features.

$$\frac{(v \vee a_1 \vee \ldots \vee a_n) \quad (\neg v \vee b_1 \vee \ldots \vee b_m)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \tag{3}$$

BDD Projection. *Binary decision diagrams* (BDDs) can be used to slice a variable as one can disjunct two BDDs and apply a conditioning (i.e., set variables to $\top$ or $\bot$) in polynomial size and time [2, 20]. Hence, you can apply existential quantification by building the BDD $\beta_{slice} = (\beta \wedge v) \vee (\beta \wedge \neg v)$ from the original BDD β. Following this procedure, the BDD needs to be duplicated and disjuncted for every sliced variable which increases the computational effort per variable. In particular, the BDD may grow exponentially in the worst case as it is duplicated for every variable.

3.4 d-DNNF

A *d-DNNF* is a negation normal form that is also *deterministic* and *decomposable* [18]. Deterministic means that for all disjunctions each pair of disjuncts shares no satisfying assignments (i.e., $\forall d_i, d_j, i \neq j : d_i \wedge d_j \equiv \bot$). A formula is decomposable if for every conjunction the conjuncts share no variables. The combination of these properties enables linear time algorithms on the d-DNNF (w.r.t. its size) for SAT, #SAT, and enumeration of satisfying assignments [20]. Various feature-model analyses depend on solving potentially many of those computational problems [12, 66, 71, 73]. Hence, the initial effort of *compiling* to d-DNNF can often be amortized and yield runtime benefits [12, 66, 73]. Often, *decision-DNNFs* are considered which are a *specialization* of d-DNNFs. Here, each disjunction is a *decision node* which has the form $d_v = (v \wedge \phi) \vee (\bar{v} \wedge \psi)$. In that example, v is the *decision variable* of the decision node. A decision node is by definition always deterministic. Since decision-DNNF are a special case of d-DNNFs, the same computational problems can be solved in linear time as for d-DNNFs.

d-DNNF Compilation. With current d-DNNF compilers [18, 45, 52], the d-DNNF is built by using the *trace* of an *exhaustive DPLL traversal*. All available compilers use a *conjunctive normal form* (CNF) [18, 45, 52] as input. Figure 2 shows the trace for a traversal on the CNF $(a \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee b \vee c \vee d)$. During the iterative branching process, remaining variables are assigned true and false in one of the branches, respectively, until every clause is satisfied. For instance, starting with the original formula assigning a (left branch in Figure 2) results in the formula $b \vee \bar{c} \vee d$ which is then used for further branching.

A common optimization is using *connected components*. A connected component is a subset S of the clauses C where no variable $v \in S$ is part of $C \setminus S$. Consequentially, each connected component does not influence the other connected components and the branching can be performed separately for each component. An example can be seen in Figure 2 with $\bar{b} \wedge c$. Here, we have two connected components $\{\bar{b}\}$ and $\{c\}$.

The trace depicted in Figure 2 can be used to construct a d-DNNF. The resulting d-DNNF is shown in Figure 3. Every time the algorithm branches on a variable v, a disjunction is created $(v \wedge \phi) \vee (\bar{v} \wedge \psi)$, which is essentially a decision node on v. As every

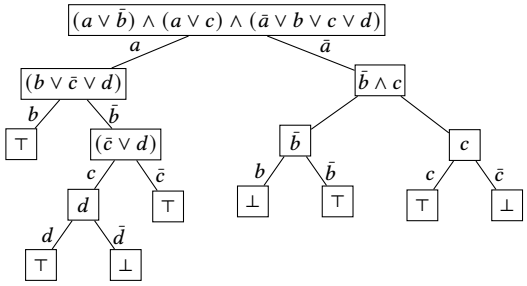


Figure 2: Exhaustive DPLL Trace

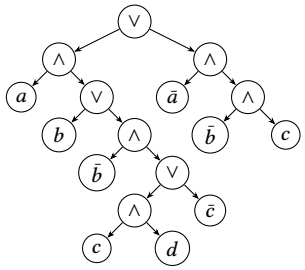


Figure 3: d-DNNF Constructed from Figure 2

solution in the left branch contains a and every solution on the right branch does not, the disjunction is deterministic by construction. When the conjunction is split into different connected components, we add a conjunction in our d-DNNF which is decomposable by construction. In the remainder of this work, we call a d-DNNF compilation that keeps all variables as *complete*, and a compilation that slices variables as *projected*.

4 PROJECTED D-DNNF COMPILATION

In this work, we introduce *projected d-DNNF compilation*, a novel way to acquire projected d-DNNFs in one step. Figure 4 shows the idea of our approach. Instead of subsequently using slicing and then d-DNNF compilation, our proposal combines them into one step, directly producing a projected d-DNNF. We realize this idea by adapting regular d-DNNF compilation to only consider projected variables. Instead of using the trace of a full exhaustive DPLL traversal as shown in Figure 2, we discard sliced variables during the traversal. In theory, our approach can be applied to every propositional formula. The d-DNNF properties (determinism and decomposability) are not restrictions on the input formula but rather guarantees by our procedure on the resulting knowledge compilation artifact. The only potential limitation for propositional instances is scalability as the underlying problem is computationally complex. In this section, we first explain the base algorithm and then discuss an intuition for its correctness.

4.1 Base Algorithm

Algorithm 1 shows our base algorithm to perform projected d-DNNF compilation which is an exhaustive DPLL traversal with the following adaptations. First, we only branch on projected variables (cf. line 7), as otherwise decisions on the sliced variables may have

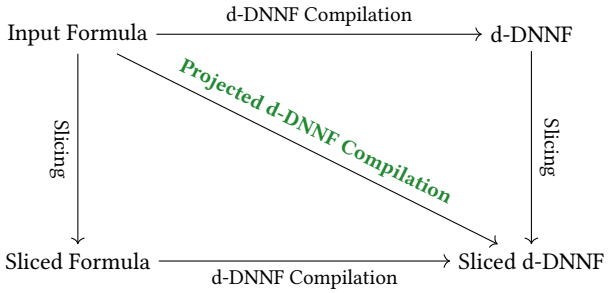


Figure 4: Commuting Diagram Showcasing Projected d-DNNF Compilation

Algorithm 1 Projected d-DNNF Compilation

Input: CNF F, set of projected variables pV
Output: projected d-DNNF

```
1:  procedure PD-DNNF(F, pV)
2:      if !SAT(F) then return ⊥
3:      subNodes ← ∅
4:      for φ in connectedComponents(F) do
5:          if vars(φ) ∩ pV ≠ ∅ then        ▷ Includes projected vars
6:              d_v ← nextDecisionVariable(φ) ▷ d_v ∈ vars(F) ∩ pV
7:              D_φ ← PD-DNNF(φ ∧ d_v) ∨ PD-DNNF(φ ∧ d̄_v)
8:              subNodes ← subNodes ∪ D_φ
9:      if |subNodes| = 0 then return ⊤      ▷ All variables sliced
10:     return ⋀_{D∈subNodes} D
```

transitive impact on the satisfying assignments of projected variables. Second, we ignore connected components that only consist of sliced variables (cf. line 5). The main procedure Algorithm 1 uses recursion to compute a partial d-DNNF for the current CNF. If the CNF is not satisfiable, we always return a $\bot$-node for the current CNF. Otherwise, we divide the CNF into the possible connected components (possibly only one) which are sets of clauses that share no variables with any other connected component (cf. Section 3.4). Then, we discard each component that includes only sliced variables. Note that each connected component is satisfiable, as we would otherwise directly return $\bot$ (cf. line 2).

For each component including at least one projected variable, we branch on one of the projected variables. We then recursively compute the partial d-DNNFs for both branches. A disjunction over the d-DNNFs resulting from both branches (cf. line 8) is semantically equivalent to the currently considered connected component as $\phi \equiv (\phi \wedge d_v) \vee (\phi \wedge d_{\bar{v}})$. A conjunction over these disjunctions is semantically equivalent to the input CNF F and is the result of the procedure `pd-DNNF`. Note that selecting promising variables for branching is a vital aspect for performance. We discuss our selection heuristics in detail in Section 5.

Figure 5 shows the impact of the adaptations on the DPLL trace of our running example. Here, we slice (i.e., remove) the variables c and d. In the first step, we have exactly one connected component and, thus, branch on one of the variables. Here, we just need to ensure that we branch on one of the projected variables. In the right

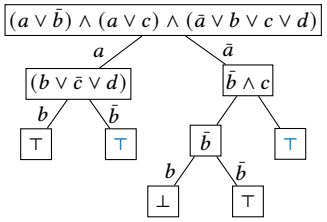


Figure 5: Projected Exhaustive DPLL Trace for Slicing c & d

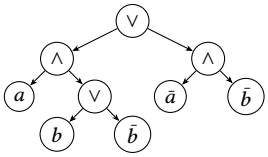


Figure 6: Projected d-DNNF Constructed from Figure 5

branch where we set $\bar{a}$, decision propagation leads to the formula $\bar{b} \wedge c$ which consists of two connected components. The right one (c) only contains sliced variables, and we replace it with $\top$ since it is satisfiable. Similarly, on the left side, after we branch on $\bar{b}$, we only have one connected component $\bar{c} \vee d$, which only contains sliced variables. Instead of further branching on $(\bar{c} \vee d)$ and c as done in Figure 2, we check its satisfiability and replace it with $\top$. We can apply the same rules as shown in Section 3.4 to convert the trace (i.e., Figure 5) to a d-DNNF. The resulting projected d-DNNF can be seen in Figure 6.

4.2 Correctness

In the following, we provide an intuition why Algorithm 1 computes a correctly projected d-DNNF. To this end, we assume that the regular d-DNNF compilation [45] shown in Section 3.4 is correct. By construction, just as regular compilation, our algorithm guarantees determinism for every disjunction and decomposability for every conjunction. It remains to show that the resulting projected d-DNNF D' is indeed a slice of the full d-DNNF D following the definition in Section 3.2. In particular, we discuss why (1) every satisfying assignment of D is satisfiable for D', when reduced to the variables in D' and (2) for every satisfying assignment of D', there is an extension with sliced variables that is satisfiable. There are two main differences between the respective base algorithms for regular and projected d-DNNF compilation: further restrictions on the variable ordering and the handling of connected components that only contain sliced variables. Since variable orderings used for projected compilation are always valid for a regular d-DNNF compilation, we assume that the same variable orderings are used for simplicity. Note that the algorithm should always produce semantically equivalent d-DNNFs for different variable orderings. Consider the current subformula $F = \phi_1 \wedge \ldots \wedge \phi_s \wedge \ldots \wedge \phi_z$ with ϕ_s being a connected component with only sliced variables. If $\text{SAT}(\phi_s) \equiv \top$, we have $D'_F \equiv \phi_1 \wedge \ldots \wedge \top \wedge \ldots \wedge \phi_z$ and $D_F \equiv \phi_1 \wedge \ldots \wedge \phi_s \wedge \ldots \wedge \phi_z$. It follows that all variables appearing in the connected components $\forall i \neq s : \phi_i$ can be equally assigned for D'_F and D_F. Hence, for an assignment satisfying for D_F, we can remove the variables from

ϕ_s to derive a satisfying assignment for D'_F, which fulfills (1). For every assignment in D'_F, we can use any satisfying assignment on the variables from ϕ_s to find a satisfying assignment for D_F, which fulfills (2). If SAT(ϕ_s) $\equiv \bot$, we have $D'_F \equiv \bot \equiv D_F$ and, thus, the same empty set of satisfying assignments which fulfills both conditions (1) and (2). Overall, it follows that our handling of connected components for slicing indeed produces a slice of the d-DNNF resulting from compilation without the adaptation. Since the correctness of the concrete implementation and employed optimizations is also relevant for usage in practice, we also study the correctness empirically in Section 6.

5 OPTIMIZATIONS

The performance of common reasoning engines, such as SAT [9] solvers and #SAT [14, 67, 76] solvers, and knowledge compilers [18, 19, 45, 52] heavily rely on heuristics. Even though solvers are based on the same base algorithms (e.g., DPLL or CDCL [48]), there are substantial differences regarding their scalability for feature models [58, 70]. In this section, we present the different heuristics we apply for optimizing the base approach presented in Section 4.

5.1 Preprocessings

As initial step, we perform different preprocessings that simplify the formula to accelerate the actual compilation. Preprocessings are a common strategy employed in reasoning engines [43–45, 69, 76]. Here, the formula is adapted so that the main algorithm still produces a semantically equivalent result but potentially with less resources, such as runtime or memory. For projected d-DNNF compilation, we collected various preprocessings from projected model counters [43, 67, 69]. However, several popular preprocessings are not trivially applicable for projected d-DNNF compilation as they only preserve the model count but not equivalence [42, 43]. Consequentially, subsequent analyses on the d-DNNF may produce faulty results. In the following, we discuss the preprocessings we apply and adaptations we made to prevent information loss.

Equivalent Literal Substitution. Two literals are equivalent if they evaluate to the same truth value ($\top$ or $\bot$) for every satisfying assignment [46]. For a set of equivalent literals $\{l_1, \ldots, l_n\}$, we can substitute each literal with a representative from the set. Note that the literals can be either positive or negative. For instance, we can replace each of $l_2, \ldots l_n$ with l_1 and each of $\bar{l_2}, \ldots \bar{l_n}$ with $\bar{l_1}$. In order to reason about all features on the compiled d-DNNF, we however need to keep track of substituted literals. We realize this by storing the equivalence sets and replacing the representative literal l_1 with $l_1 \wedge l_2 \wedge \ldots \wedge l_n$ and $\bar{l_1}$ with $\bar{l_1} \wedge \bar{l_2} \wedge \ldots \wedge \bar{l_n}$ in the compiled d-DNNF. Without those replacements, we would lose the information of the substituted equivalent literals and could not consider the corresponding variables in analyses on the d-DNNF. For example, we could not query how many satisfying assignments include l_2.

Partial Resolution. As described in Section 3.3, logical resolution can be used to slice variables. However, in the worst case we have a quadratic increase of clauses *per variable* to be sliced. The idea of *partial resolution*, which is used in the projected model counter GPMC,[3] is to resolve only sliced variables that do not increase the number of clauses or the *tree-width* [24] of the formula. Projected model counting has exponential time complexity w.r.t. tree-width, which is a metric that describes how close a graph is to a tree [26]. To measure the tree-width, the formula is represented as undirected graph where each variable is a node and there is an edge between those nodes if both variables appear together in a clause. After applying partial resolution, we use the simplified formula with some variables already sliced for the projected d-DNNF compilation.

Backbone Propagation. The *backbone* of a propositional formula F describes the literals that need to be satisfied in order for the formula to be satisfiable [34]. Formally, a literal l is part of the backbone if $\bar{l} \wedge F \equiv \bot$. Hence, the formula F can be simplified by replacing all l with $\top$ and $\bar{l}$ with $\bot$ and then perform decision propagation. With decision propagation, clauses that contain a $\top$ are removed as they are satisfied and $\bot$ is removed from clauses. Then, if a new unit clause was produced, the respective variable is again substituted with $\top$ and $\bot$ until a fixpoint is reached. Again, for projected d-DNNF compilation, we need to keep track of those variables to produce a complete d-DNNF. After the compilation, we attach the literals to the d-DNNF D that are part of the backbone B as $D \wedge \bigwedge_{l \in B} l$. Note that this top level conjunction is always decomposable as the variable backbones are not part of D as they were temporarily discarded during the compilation.

5.2 Weighted Hypergraph Partitioning

Connected components (i.e., sets of clauses that do not share variables) can be processed separately without worrying about interdependencies [6, 76]. Consequentially, a strategy often followed by #SAT solvers [6, 59, 76] and d-DNNF compilers [45] is finding *variable orderings* that split the formula to multiple connected components within only few decisions.

The best performing d-DNNF compiler for feature models [71] d4 [45] uses *hypergraph partitioning* for deriving a variable order. Here, the CNF is represented as a hypergraph with each node being a clause. The hyperedges correspond to one variable each and indicate which clauses the variable occurs in. With hypergraph partitioning, we aim to find a *cut* which is a set of variables/edges that, when removed, split the hypergraph in two partitions with similar size. After deciding all variables from the cut in branching, our CNF will also be split into distinct connected components.

Regular hypergraph partitioning, such as used in d4, is not applicable for projected d-DNNF compilation. First, we cannot remove sliced variables for the partitioning as assigning them during branching may have a transitive impact on the projected variables. Second, we cannot ignore their dependencies as this could result in components that share sliced variables. Then, the components are not independent and, thus, cannot be processed separately.

To prevent the partitioning from using sliced variables while still considering their dependencies, we propose to use *weighted hypergraph partitioning* [61]. With that generalization of hypergraph partitioning, each edge is assigned a weight (or cost). The algorithm then tries to find a partition with a low cost of removed edges. We set the weight of projected variables to a low constant and the sliced variables to a very high constant. With high enough weights, the sliced variables are not selected but also not disregarded if they connect partitions.

[3] https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC

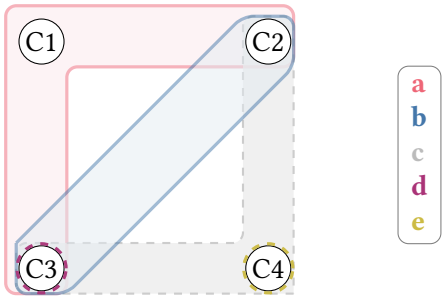


Figure 7: Weighted Hypergraph

Figure 7 shows an example for a weighted hypergraph for an adapted variant of our running example $F' = (a \vee \bar{b}) \wedge (a \vee c) \wedge (\bar{a} \vee b \vee c \vee d) \wedge (c \vee e)$. The four nodes C1-C4 represent the clauses of F'. The edges (or boxes) represent the variables. For instances, the red box represents the variable a, which is part of the clauses C1, C2, and C3. Projected variables are represented by solid lines (low weights) and sliced variables by dashed lines (high weights). In our example Figure 7, both a and c would partition the formula into two parts, respectively. However, due to the weighting the partitioning algorithm does not consider c as cutting variable. Removing a partitions the graph into two components {C1} and {C2, C3, C4} which then can be separately compiled to d-DNNF. Consequentially, our procedure would use a first in the variable ordering for branching. Note that it may be necessary to remove multiple variables before achieving a partitioning. Just as regular d4, we use hypergraph partitioning at the start of the compilation procedure and also dynamically on demand during the compilation.

As an optimization, we can use *equivalent variables* to also simplify the hypergraph. In the hypergraph, we replace the set of edges corresponding to equivalent variables with one edge. This comes with two main advantages: First, the hypergraph contains fewer edges which simplifies partitioning. Second, if the set of equivalent variables contains at least one projected variable, we can assign a low weight to this variable, as all sliced variables are implicitly decided by branching on the projected variable. Note that the initial hypergraph does not contain equivalent variables if we use our preprocessing to substitute equivalent literals. However, simpifying the hypergraph with equivalent variables is applicable for dynamic invocations of the partitioning or if the preprocessing is turned off.

5.3 Other Optimizations

Variable Ordering. With weighted hypergraph partitioning, we identify a set of variables to prioritize during branching. Still, the runtimes of the compiler depend on specific *variable orderings* within this set. For our compiler, we use a slightly adapted version of the commonly used [45, 67] *variable state aware decaying sum (VSADS)* [59]. VSADS combines two heuristics that prioritize (1) variables appearing in many clauses (i.e., DLCS) and (2) variables that satisfy recently identified conflict clauses (i.e., VSIDS) [59]. For our projected d-DNNF compilation, we adapt VSADS to only consider projected variables.

Pure Literal Elimination. A literal l is pure if $\bar{l}$ does not occur in the formula [35]. A pure literal can be trivially sliced during compilation as we can just assume $\top$ for positive or $\bot$ for negative literal as assignment. As no clause contains $\bar{l}$, the assumed assignment does not falsify any satisfying assignment over projected variables.

5.4 Tool Support

We realize the main concept and all the listed heuristics in our tool pd4, which is an extension of the popular d-DNNF compiler d4. We use d4 as recent work showed that d4 performs better on feature models than other available d-DNNF compilers [71]. pd4 takes CNFs in DIMACS format as input with an additional header for specifying the projected variables. The header follows the syntax `c p show v1 v2 v3 ...` with `v1`–`v3` being the variables to project. This header is also used in the projected model counting track of the Model Counting Competition [25]. pd4 is available at GitHub.[4]

6 EVALUATION

In our empirical evaluation, we examine the performance of our proposal pd4 compared to state-of-the-art tools. A replication package including solvers, data (excluding confidential models), and scripts to reproduce the experiments is publicly available.[5]

6.1 Research Questions

RQ1 How does the performance of projected d-DNNF compilation compare to slicing combined with regular compilation?

For **RQ1**, we aim to compare performance of the current state-of-the-art for acquiring a projected d-DNNF. As we are the first to introduce projected d-DNNF compilation, we compare our prototype to a two-step approach that subsequently performs slicing and then d-DNNF compilation. For the comparison of performance, we consider overall runtime and size of the produced d-DNNFs. As many subsequent analyses base on traversals of the d-DNNF [66, 73], the size is also relevant for runtime of feature-model analyses when using the compiled d-DNNF.

RQ2 How does the performance of projected d-DNNF compilation compare to projected model counting?

With **RQ2**, we compare the performance of projected d-DNNF compilation to closely related state-of-the-art tools, namely projected model counters. While projected model counters only compute the number of satisfying assignments instead of a reusable d-DNNF, they rely on the same base technique as our approach (i.e., exhaustive DPLL traversal).

RQ3 How do optimizations impact the performance of pD4?

For **RQ3**, we perform an ablation study to examine the impact of the central optimizations we employ for our approach. In particular, we analyze the impact of the preprocessings and weighted hypergraph partitioning presented in Section 5.

6.2 Experiment Design

Subject Systems. We use three datasets shown in Table 1 for comparison to increase external validity. First, we use twelve real feature models with four slices each that are actually applied in

[4] https://github.com/SoftVarE-Group/pd4
[5] https://doi.org/10.5281/zenodo.13752743

Table 1: Subject Systems

Track	Instances	Variables	Clauses
Industrial Models & Slices	12 · 4	376–3,286	1,015–51,450
Literature Models & Random Slices	49 · 100	17–62,482	16–350,221
Competition Instances	200	100–3,423,788	545–6,163,392

Table 2: Evaluated Tools

Name	Type	Origin
pd4	Projected d-DNNF Compiler	This work
Slicing FeatureIDE (v3.10)	Slicer via Resolution	[39]
d4 (v2)	d-DNNF Compiler	[45]
GPMC (v1.1.1)	Projected Model Counter	[25]
d4-pmc (v2)	Projected Model Counter	[45]
arjun (MCC Version)	Projected Model Counter	[67, 69]

practice from our industry partner. The feature models represent configurable systems from the automotive domain and each slice represents a part of the feature model for different disciplines. Second, we consider publicly available real-world feature models from the literature and generate random sets of features to slice from them. Here, we generate 100 slices for every feature model with each slice removing between 10% and 80% of the variables. Third, we compare the performances on instances of the Projected Model Counting Track from the Model Counting Competition 2022.[6]

Evaluated Tools. Table 2 shows the tools we consider for comparison in our experiments. First, we evaluate our projected d-DNNF compiler `pd4`. Second, we consider a combination of slicing as performed by FeatureIDE [39] and the regular `d4` [45] compiler as baseline for acquiring a projected d-DNNF. Third, we compare the performance to the best performing projected model counters from the Projected Model Counting Track from the Model Counting Competition (MCC) 2022 and 2023, namely `GPMC`,[7] `d4-pmc`,[8] and `arjun` [69]. `arjun` is built on top of the solver `sharpSAT` [76].

Experiment Execution. For the experiments, we preprocessed every input instance as CNF in DIMACS format [10] with additional meta information indicating the variables that are part of the projection. We then execute each tool listed in Table 2 on every CNF projection representing a feature model. For each projection, we set a timeout of 10 minutes as (1) there were no substantial changes to the results by increasing the timeout in preliminary experiments and (2) the high number of instances are already computationally expensive with 10 minutes of timeout. For the models from the model counting competition, we use the same timeout as in the competition which is 60 minutes per instance. We repeat each measurement three times to reduce the effects of computational bias. If not stated otherwise, we present the median of those three repetitions in this section. For the combination of slicing and regular `d4`, we execute slicing first and then invoke `d4` on the resulting sliced CNF. To reduce the demand of computational resources, we perform the ablation study on the different optimization variants only on the feature models where we have actual slices from industry available, because these are assumed to be more realistic for the research focus (i.e., feature-model slicing). To examine the correctness of tools, we compared resulting model counts. For the d-DNNFs approaches, we used `ddnnife` [73] to compute model counts on the compiled d-DNNFs.

[6] https://mccompetition.org/past_iterations
[7] https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC
[8] https://github.com/crillab/d4v2

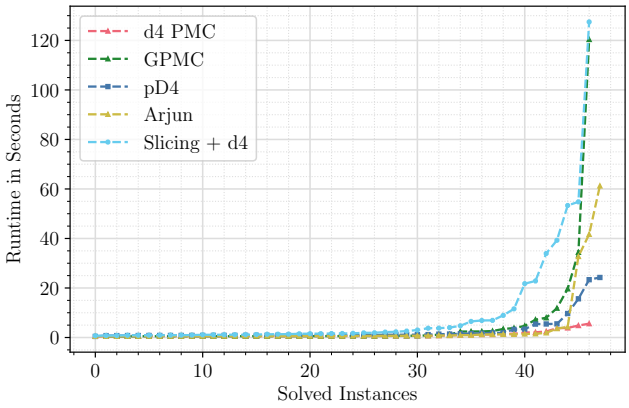
6.3 Results

Industry Models & Slices. Figure 8a shows a cactus plot indicating runtime of the evaluated tools on the industry models with real slices. The x-axis shows the number of solved instances and y-axis the runtime in seconds. Note that for each tool the runtimes are sorted individually on the x-axis. In general, the further to the right a line for a tool is drawn, the more instances have been solved by this tool. `arjun` and our projected d-DNNF compiler `pd4` are able to solve to the most instances (47). For the 47 instances, `pd4` requires 136 seconds and `Arjun` 176 seconds. Projected model counting with `d4` (`d4-pmc`) only solves 46 instances but for the instances with respective lowest runtimes, `d4-pmc` is faster with 52.0 seconds while `pd4` requires 112 seconds. The combination of slicing with FeatureIDE and d-DNNF compilation scales to 46 models as well but requires 454 seconds. For instances solved by both `pd4` and slicing, `pd4` is 4.05 overall times faster. All d-DNNFs compiled by `pd4` during our experiments produced the same model counts as all projected model counters.

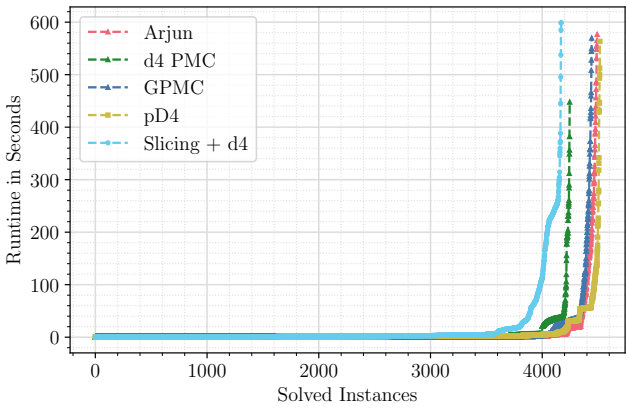
Literature Models & Random Slices. Figure 8b shows a cactus plot for the runtimes on the feature models from the literature and artificial slices. Our tool `pd4` solved the most instances (4,516) followed by `Arjun` (4,492). While `pd4` required 8.19 hours for the 4,516 instances, `Arjun` needed 8.86 hours for 4,492 instances. The combination of slicing and `d4` successfully terminated for the fewest models (4,170). Note that the literature dataset contains various feature models with few features and constraints. Hence, every considered tool, even slicing, scales for most instances. In particular, every solver required less than one second each for the 3,000 instances inducing the respective shortest runtimes. For instances solved by both `pd4` and slicing, `pd4` is overall 9.25 times faster. We are also able to compile slices for feature models, for which regular d-DNNF compilation does not scale with available compilers [71]. For instance, `pd4` successfully produced sliced d-DNNF for 96 of 100 slices for Linux (v2.6.33.3), but compiling to complete d-DNNF does not scale [71].

Competition Instances. Figure 8c shows a cactus plot for the runtimes on formulas of the Projected Model Counting track of the Model Counting Competition 2022. Again, our tool `pd4` solves the highest number of instances (159) followed by `GPMC` (155). `pd4` solves those 159 instances in 23 hours while `GPMC` requires 26 hours for 155. The combination of slicing and `d4` successfully terminates for the fewest instances (52). For instances solved by both `pd4` and slicing, `pd4` is overall 22.0 times faster.

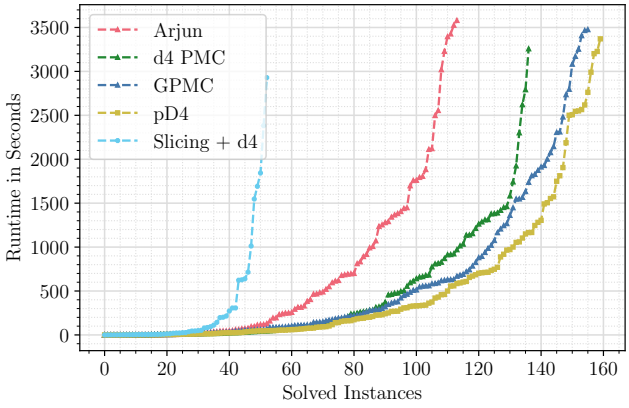
d-DNNF Sizes. Figure 9 shows the number of nodes for the d-DNNFs produced by `pd4` and slicing combined with regular `d4`. The plots only include instances solved by both solvers. A marker below the blue line indicates that `pd4` computed a smaller d-DNNF.



(a) Evaluated Tools on Industry Models with Real Slices



(b) Evaluated Tools on Models from Literature with Generated Slices



(c) Evaluated Tools on Instances from the MC2022

Figure 8: Runtimes of Evaluated Tools on Different Datasets

Overall, the sizes of the produced d-DNNFs are comparable. For feature models, pd4 tends to produce smaller d-DNNFs (for 61.7% of the instances). On the literature models, pd4 produces on average 5.07 times smaller d-DNNFs. On the industrial models, pd4 produces on average 25.1% smaller d-DNNFs. For the competition instances, slicing was able to compute considerably fewer d-DNNFs (53 vs 160), but slicing produces a smaller d-DNNF than pd4 in 78.8% of the cases for those successfully evaluated by both.

pd4 Optimizations. Figure 10 shows the impact of the different optimizations as a cactus plot for the industry instances. Turning off hypergraph partitioning has a smaller impact on the performance than preprocessing, but is still considerable. While the same number of instances are solved with and without partitioning, the overall runtimes for the solved instances are 136 seconds and 734 seconds (81.5% decrease), respectively. Without preprocessing, pd4 only solves 35 of the instances independently of using partitioning.

6.4 Discussion

RQ1 *How does the performance of projected d-DNNF compilation compare to slicing combined with regular compilation?* On all three datasets, the projected d-DNNF compilation with our tool pd4 substantially outperformed the subsequent execution of slicing and regular compilation. pd4 scales for far more instances than slicing followed by d-DNNF compilation. As slicing followed by invoking a regular d-DNNF compiler appears to be the only alternative that produces a projected d-DNNF, our results suggest that pd4 is the most promising approach to derive projected d-DNNFs. A further potential benefit is that we were able to compile projected d-DNNFs for which we could neither compile a complete d-DNNF nor perform slicing. The sizes of compiled d-DNNFs were comparable for both slicing with regular d4 and pd4, but pd4 generally produces smaller d-DNNFs for the feature-model instances.

RQ2 *How does the performance of projected d-DNNF compilation compare to projected model counting?* During all experiments, pd4 outperforms all projected model counters even though pd4 produces a d-DNNF for further reuse (e.g., [12, 66, 73]) instead of just the model count (i.e., a single number). Over the three datasets, pd4 scales for more instances than any projected model counter. For instances solved by all tools, pd4 is typically competitive with the projected model counters, but still sometimes slower. The largest observed difference is on the feature models from the literature where pd4 is 3.92 times slower than `Arjun` for the easier instances. Even for those cases, pd4 has considerable benefits as it returns both the model count and the d-DNNF which can be reused. The runtime advantages of pd4 over slicing and projected model counters on instances from the model counting competition indicate that pd4 is also promising for application domains beyond product lines and even if only the model count is needed.

RQ3 *How do optimizations impact the performance of pD4?* The performed ablation study strongly indicates that the employed optimizations positively impact the runtime of pd4. Weighted hypergraph partitioning substantially reduces runtimes, but pd4 still scaled for the same models without it. The preprocessings are crucial for the scalability, as pd4 scaled to considerably fewer instances without them.

6.5 Threats to Validity

External Validity of Slices. For the feature models from the literature, we have no information available on practically relevant slices. The randomly generated slices may not be representative

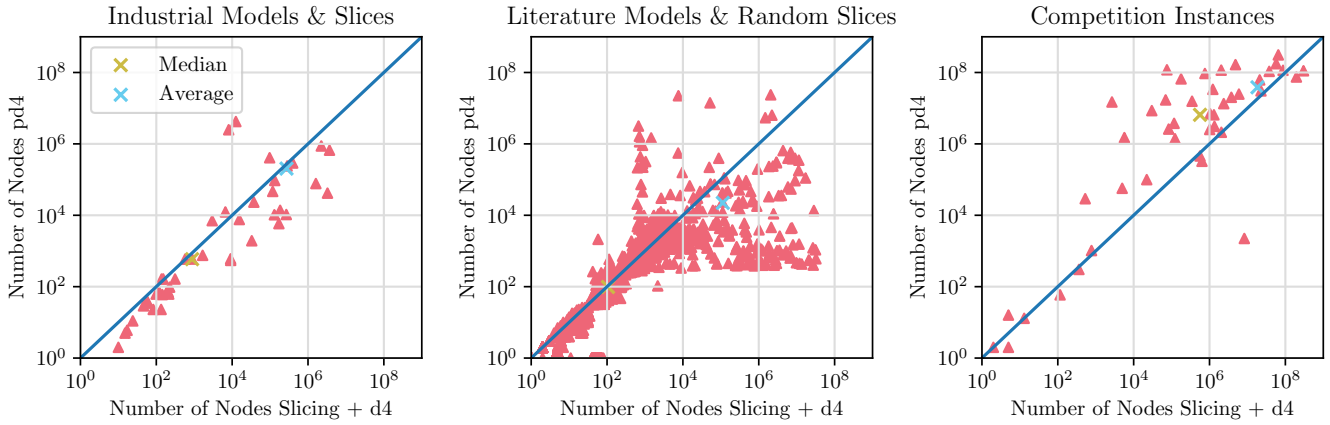


Figure 9: Number of Nodes for Produced d-DNNFs

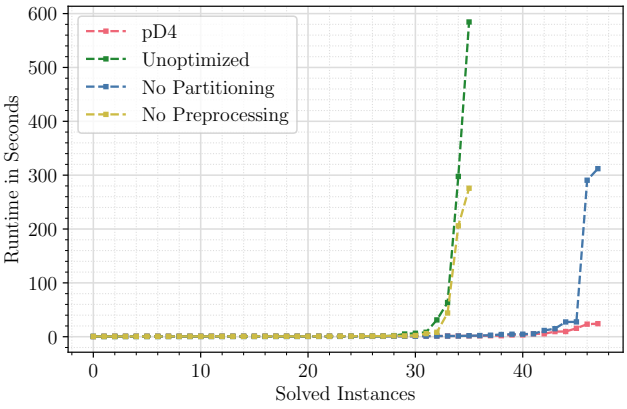


Figure 10: Impact of pD4 Optimizations

to slices needed in practice. The main issue is the availability of real-world slices on feature models. For the models from literature, we aim to mitigate the issue by acquiring a reasonable coverage with many slices of various sizes. Furthermore, we compare each tool on slices from practice provided by our industry partner and available projections from the model counting competition.

External Validity of Feature Models. Our selection of input instances may not be transferrable for all practically relevant feature models. To mitigate the issue, we collected feature models from various publications, domains, and with wide ranges of structural properties (e.g., 17–62,482 variables and 16–350,221 clauses). Furthermore, we examined the performance of our tool on instances beyond feature models, where `pd4` also performed well.

External Validity of Solver Parameterizations. Several tools used in our experiments offer parameters to control performance-relevant behavior of the algorithms. During our experiments, we invoked each solver with default parameters. Using other parameters may influence the performance of the different tools. However, examining different parameters vastly increases the complexity of the already computationally demanding evaluation due to combinatorial explosion of the parameters. Furthermore, we expect default parameters are a decent indicator for usage of these tools in practice.

Internal Validity. The measured runtimes of tools may differ between individual runs for the same input instance due to *computational bias*. To the best of our knowledge, each tool behaves deterministicly which reduces the impact of *random effects*. For more accurracy on the measured runtimes, we performed three repetitions per measurement. Furthermore, we argue that the multitude of input instances (5,148 per tool) also compensates for potential bias due to individual runs.

7 RELATED WORK

Projected Model Counting. Projected model counters compute the number of satisfying assignments induced by a subset of variables [26]. With our procedure, we essentially compute a projected d-DNNF from the trace of an exhaustive DPLL traversal which are also used by the projected model counters. Hence, the techniques are closely related. However, to the best of our knowledge, none of the available projected model counters supports producing a d-DNNF as output. Furthermore, many projected model counters use preprocessings that preserve the model count but not equivalence [67, 69]. Hence, the traces from the performed DPLL traversals cannot be directly used for a correct and complete d-DNNF. While we adapted some preprocessings to work for projected d-DNNF compilation (cf. Section 5), others, such as B+E [43] would require substantial computational effort to preserve equivalence.

d-DNNF Compilation. Three d-DNNF compilers are typically considered in the literature, namely `d4` [45], `dSharp` [52], and `c2d` [18]. The latest version of `d4`[9] supports projected model counting. However, to the best of our knowledge, no d-DNNF compiler supports compilation to a projected d-DNNF. In general, projected knowledge compilation has not been considered before in any domain. All listed compilers should be adaptable with our approach, we extended `d4` as it performed best for feature models in recent work [71].

[9] https://github.com/crillab/d4v2

d-DNNF Projection. It is possible to perform a projection on a given d-DNNF within linear runtime with respect to the number of d-DNNF nodes [3]. However, during the projection process the determinism is not preserved. The resulting format DNNF supports fewer linear-time queries for analyses [20]. For instance, counting in linear time complexity, which is one of the major selling points of d-DNNFs [20, 66, 73], is not possible a on DNNF. Aziz et al. [3] propose to (1) compile to d-DNNF with regular compilation, (2) perform a projection on that d-DNNF, (3) translate the resulting DNNF to CNF, and (4) compile this CNF to d-DNNF again. They also realized this in a prototype d2c, which we excluded for our experiments since it produced faulty results for the majority of instances in preliminary experiments. Also, this procedure requires to compute a d-DNNF over the entire feature model first, which is often expensive or may not even scale to complex models (cf., Figure 8b), such as the Linux kernel [71].

Feature-Model Slicing. Acher et al. [2] proposed slicing on binary decision diagrams. Slicing one variable can be performed in linear time and space with respect to the size of the BDD [20]. However, this procedure requires compiling to BDD first, which has been found to not scale for many feature models in recent work [32, 71]. Also, BDD compilation scales substantially worse than d-DNNF compilation for feature models [71]. Thüm et al. [75] use existential quantification to eliminate abstract (i.e., not relevant for the implementation) features from configuration to derive program variants. Krieter et al. [39] compare different slicing algorithms based on logical resolution. They integrated the best performing variant in FeatureIDE, which is also the variant we used in our empirical evaluation which showed that projected d-DNNF compilation scales to substantially more instances.

Feature-Model Analysis on d-DNNFs. The popularity of using d-DNNFs for feature-model analysis increased in the last ten years [12, 66, 73]. In recent work, we reduce a variety of feature-model analyses to d-DNNF counting operations. Sharma et al. [66] use d-DNNFs to compute uniform random samples. Bourhis et al. [12] compare using different one-time computation tools to d-DNNFs for counting, enumerating, and optimal configurations. The listed work operate on an existing d-DNNF while we propose the first algorithm to compile to projected d-DNNF. The plethora of analyses on d-DNNFs further motivates our work, as the proposed algorithms can also be applied to the projected d-DNNFs.

Projected Reasoning Beyond Product Lines. There are various relevant analyses in different application domains based on automated reasoning targeting projections [69]. Such analyses are mostly reduced to satisfiability, counting, and sampling (i.e., enumeration) [69], which all can be performed in polynomial runtime complexity on d-DNNFs. Duenas-Osorio et al. [22] use projected model counting for evaluating *network reliability*. Zawadzki et al. [77] employ projected model counting for *planning problems*. Teuber and Weigl [74] evaluate *software reliability* using projected model counting. In *computational biology*, Sashittal and El-Kebir [60] employ enumeration and counting on projected formulas representing transmission histories of disease outbreaks. Baluta et al. [4] verify different security-relevant properties of neural networks using repetitive invocations of projected model counting. Since the listed applications rely on operations that can be quickly performed on d-DNNFs, our procedure is potentially beneficial there, especially considering the strong performance on non-product-line instances in our evaluation.

d-DNNF Applications Beyond Product Lines. Various computationally complex problems can be solved on d-DNNFs in polynomial time with respect to their size [20]. Checking satisfiability and model counting, which are both relevant problems in many application domains [25, 27], can be computed even in linear time complexity. Hence, many application domains can benefit from the application of d-DNNFs. In addition to product lines [12, 66, 73], d-DNNFs have been applied in conformant planning [56], recommender systems [11], explainable machine learning [33], sequential diagnosis [68], and inference from bayesian networks [15]. In these domains, which already use d-DNNFs, our proposal may also be useful in the future if applications on projections emerge.

8 CONCLUSION

Typically, only parts of a configurable system are relevant for particular stakeholders. Slicing enables considering only a subset of features while preserving transitive dependencies. With our proposal of projected d-DNNF compilation, we tackle two issues of existing slicing approaches: the scalability of the slicing process and the runtime of subsequent analyses on the sliced feature model.

Our empirical evaluation strongly indicates that our prototype outperforms the state of the art of slicing and also even closely related projected model counters. Overall, projected d-DNNF compilation and our prototype pd4 seem to be a promising solution for slicing, when subsequent analyses are required. Projected d-DNNF compilation may also be beneficial for other domains as our prototype also scales to more instances beyond feature models from the model counting competition. Since we found feature models for which we cannot compile a complete d-DNNF but a sliced one, this may enable additional analyses for those complex models.

ACKNOWLEDGMENTS

This work is based on the master's thesis of Loth. We thank Paul Bittner, Rahel Sundermann, and Tobias Heß for their valuable feedback on our work.

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages.

[2] 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.

[3] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. 2015. #∃ SAT: Projected Model Counting. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 121–137.

[4] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 1249–1264.

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

[6] 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.

[7] 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.

[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering (TSE)* 39, 12 (2013), 1611–1640.

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

[10] Armin Biere, Marijn Heule, Hans van Maaren, and Tony Walsh. 2009. *Handbook of Satisfiability*. IOS Press.

[11] Pierre Bourhis, Laurence Duchien, Jérémie Dusart, Emmanuel Lonca, Pierre Marquis, and Clément Quinton. 2022. *Pseudo Polynomial-Time Top-k Algorithms for d-DNNF Circuits*. Technical Report. Cornell University Library.

[12] 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.

[13] Randal E. Bryant. 2018. Binary Decision Diagrams. In *Handbook of Model Checking*. Springer, 191–217.

[14] 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.

[15] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning (IJAR)* 42 (2006), 4–20.

[16] 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.

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

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

[19] Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *International Joint Conferences on Artificial Intelligence*. AAAI Press, 819–826.

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

[21] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. 2007. A Generic Solution to Software-in-the-Loop. In *IEEE Military Communications Conference (MILCOM)*. 1–6.

[22] Leonardo Duenas-Osorio, Kuldeep Meel, Roger Paredes, and Moshe Vardi. 2017. Counting-Based Reliability Estimation for Power-Transmission Grids. *Proc. Conf. on Artificial Intelligence (AAAI)* 31 (2017). https://doi.org/10.1609/aaai.v31i1.11178

[23] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 549–559.

[24] Aurélie Favier, Philippe Jégou, and Simon De Givry. 2011. Solution Counting for CSP and SAT With Large Tree-Width. *Control Systems and Computers (CSC)* 2 (2011), 4–13.

[25] 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.

[26] Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. 2018. Exploiting Treewidth for Projected Model Counting and Its Limits. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, Cham, Switzerland, 165–184.

[27] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. 1996. *Algorithms for the Satisfiability (SAT) Problem: A Survey*. Technical Report. Cincinnati University.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Xuanxiang Huang, Yacine Izza, Alexey Ignatiev, Martin Cooper, Nicholas Asher, and Joao Marques-Silva. 2022. Tractable Explanations for d-DNNF Classifiers. In *Proc. Conf. on Artificial Intelligence (AAAI)*, Vol. 36. AAAI Press, 5719–5728.

[34] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. 2015. Algorithms for Computing Backbones of Propositional Formulae. *AI Communications* 28, 2 (2015), 161–177.

[35] Jan Johannsen. 2005. The Complexity of Pure Literal Elimination. *J. Autom. Reason.* 35 (2005), 89–95.

[36] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[37] 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.

[38] Sebastian Krieter. 2020. Large-Scale T-Wise Interaction Sampling Using YASA. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 29:1–29:4.

[39] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 60–64.

[40] 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.

[41] 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.

[42] 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.

[43] Jean-Marie Lagniez, Emmanue Lonca, and Pierre Marquis. 2016. Improving Model Counting by Leveraging Definability. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 751–757.

[44] Jean-Marie Lagniez and Pierre Marquis. 2014. Preprocessing for Propositional Model Counting. *Proc. Conf. on Artificial Intelligence (AAAI)* 28, 1 (2014).

[45] 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.

[46] Yong Lai, Kuldeep S Meel, and Roland HC Yap. 2021. The Power of Literal Equivalence in Model Counting. In *Proc. Conf. on Artificial Intelligence (AAAI)*, Vol. 35. AAAI Press, 3851–3859.

[47] 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.

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

[49] 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.

[50] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

[51] 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.

[52] 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.

[53] 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.

[54] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 84–87.

[55] 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.

[56] Héctor Palacios, Blai Bonet, Adnan Darwiche, and Hector Geffner. 2005. Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *Proc. Int'l Conf. on Automated Planning and Scheduling (ICAPS)*, Vol. 5. AAAI Press, 141–150.

[57] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2016. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 12 pages.

[58] 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.

[59] Tian Sang, Paul Beame, and Henry Kautz. 2005. Heuristics for Fast Exact Model Counting. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 226–240.

[60] Palash Sashittal and Mohammed El-Kebir. 2019. *SharpTNI: Counting and Sampling Parsimonious Transmission Networks under a Weak Bottleneck*. Technical Report.

[61] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-Quality Hypergraph Partitioning. *ACM J. of Experimental Algorithmics (JEA)* 27 (2023), 1–39.

[62] Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 252–268.

[63] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.

[64] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2017. Compositional Analyses of Highly-Configurable Systems With Feature-Model Interfaces. In *Proc. Software Engineering (SE)*, Jan Jürjens and Kurt Schneider (Eds.). Gesellschaft für Informatik, 129–130.

[65] Sergio Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 2. IEEE, 201–207.

[66] 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.

[67] 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.

[68] Sajjad Siddiqi and Jinbo Huang. 2008. Probabilistic Sequential Diagnosis by Compilation. In *Proc. Int'l Symposium on Artificial Intelligence and Mathematics (ISAIM)*. AAAI Press.

[69] Mate Soos and Kuldeep S. Meel. 2022. Arjun: An Efficient Independent Support Computation Technique and its Applications to Counting and Sampling. In *Proc. Int'l Conf. on Computer-Aided Design (ICCAD)*. ACM, Article 71.

[70] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2024. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. In *Proc. Software Engineering (SE)*. Gesellschaft für Informatik. To appear.

[71] 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).

[72] 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.

[73] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *Trans. on Software Engineering and Methodology (TOSEM)* (2024). To appear.

[74] Samuel Teuber and Alexander Weigl. 2021. Quantifying Software Reliability via Model-Counting. In *Proc. Int'l Conf. on Quantitative Evaluation of Systems (QEST)*. Springer, 59–79.

[75] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 191–200.

[76] 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.

[77] Erik Peter Zawadzki, André Platzer, and Geoffrey J Gordon. 2013. A Generalization of SAT and# SAT for Robust Policy Evaluation.. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*. AAAI Press, 2583–2590.