On the Benefits of Knowledge Compilation for Feature-Model Analyses

Chico Sundermann[1*], Elias Kuiter[2*], Tobias Heß[1*], Heiko Raab[1*], Sebastian Krieter[1*] and Thomas Thüm[1*]

[1]University of Ulm, Ulm, Germany.
[2]Otto-von-Guericke University, Magdeburg, Germany.

*Corresponding author(s). E-mail(s):
chico.sundermann@uni-ulm.de; kuiter@ovgu.de;
tobias.heß@uni-ulm.de; heiko.raab@uni-ulm.de;
sebastian.krieter@uni-ulm.de; thomas.thuem@uni-ulm.de;

Abstract

Feature models are commonly used to specify the valid configurations of product lines. As industrial feature models are typically complex, researchers and practitioners employ various automated analyses to comprehend their described configuration spaces. Many of these automated analyses require that numerous complex computations are executed on the same feature model, for example by querying a SAT or #SAT solver. With knowledge compilation, feature models can be compiled in a one-time effort to a target language that enables polynomial-time queries for otherwise complex problems. In this work, we elaborate on the potential of employing knowledge compilation on feature models. First, we gather various feature-model analyses and study their computational complexity with regard to the underlying computational problem and the number of solver queries required for the respective analysis. Second, we collect knowledge-compilation target languages and map feature-model analyses to the languages that make the analysis tractable. Third, we empirically evaluate publicly available knowledge compilers to further inspect the potential benefits of knowledge-compilation target languages.

Keywords: feature modeling, knowledge compilation, SAT, #SAT

1 Introduction

Product lines are commonly used to develop, test, and evolve a family of similar products to reduce the costs compared to developing each product separately [1–5]. Each product in a product line is composed from a set of reusable *features*, which are generally shared across multiple products [3, 6]. Typically, not every combination of selected features (i.e., a *configuration*) is valid and results in a functional product [7–9]. To specify the set of valid configurations for a given product line, engineers can use *feature models* [2, 10, 11]. A feature model consists of a set of features and a set of *constraints* limiting the valid configurations [12]. For instance, including one feature may require in- or excluding another feature.

Manually keeping track of all constraints is infeasible, as industrial feature models may contain thousands of features and hundreds of thousands of constraints [7, 13]. For example, the Linux kernel (as of November 2018) contains more than 14,000 features and 60,000 constraints [14]. Automated reasoning is typically used to analyze feature models, for example to check whether a given configuration is valid (i.e., it conforms to all constraints imposed by the feature model) [2].

Many feature-model analyses are computationally expensive [7, 9, 11, 15]. The de facto standard for analyzing feature models is to translate them into propositional logic and automate the analysis with tools, such as *SAT* [11] or *#SAT* [7, 9, 16] *solvers*. Feature models require the full expressiveness of propositional formulas [17], so checking whether a feature model has at least one valid configuration (i.e., a satisfying assignment) is NP-complete [11]. Analogously, computing the number of valid configurations is #P-complete [7, 9, 18]. Even further, many feature-model analyses depend on numerous complex solver queries [2, 7, 16]. For instance, existing tools compute for each feature whether it is core (i.e., it appears in every valid configuration) or dead (i.e., it appears in no valid configuration) to detect unintentional side effects in the modeling process [19, 20]. Naively employing a SAT solver to calculate such core and dead features requires $2 \cdot n$ SAT calls (n being the number of features).

For computing feature-model analyses that require multiple solver queries, employing *knowledge compilation* seems promising [21, 22]. With knowledge compilation, an initial effort is taken to translate the propositional formula for a feature model into an artifact of a knowledge-compilation target language, such as a *binary decision diagram (BDD)* [23] or *deterministic decomposable negation normal form (d-DNNF)* [24]. In theory, such a knowledge-compilation artifact can then be used to perform repetitive computations more efficiently [25].

However, the research on leveraging knowledge compilation for feature-model analysis is still rather limited in terms of (a) considered analyses, (b) considered knowledge-compilation target languages, and (c) compiler scalability: Regarding feature-model analyses, it is well-known that some feature-model analyses require multiple complex queries [2, 9, 16, 26–28]. However, the insights on the required computations are scattered over many works

and extracting the complexity of the computations often requires interpretation of the reader [2, 11, 16, 26, 29, 30] *[ref]*. Regarding knowledge-compilation artifacts, only few knowledge-compilation target languages have been applied to compute a few feature-model analyses [22, 31]. However, many capabilities of well-known knowledge-compilation target languages [25] have not been utilized yet for feature-model analyses. For instance, while BDDs have been employed repeatedly [21, 32–34], target languages that are popular in other domains, such as SDDs [35] or d-DNNFs [24, 25], have not been used or only limited to single applications. For example, d-DNNFs have been used for enumeration of feature models (i.e., sampling [22, 36]), but have not been used for SAT or #SAT queries yet, excluding the usage of compilers as black box #SAT solvers [7, 9]. Finally, regarding compiler scalability, knowledge compilation can only be sensibly applied for feature-model analyses if the compilation of feature models to the respective artifact scales appropriately. If the compilation takes too long or does not scale at all, the benefits (i.e., efficient queries) of the knowledge-compilation artifact vanish. However, besides preliminary work on some knowledge-compilation target languages [7, 21, 22], it is still largely unknown which artifacts can be compiled from feature models in reasonable time.

In this article, we aim towards closing these gaps by elaborating on the potential of applying knowledge compilation for feature-model analysis with the following contributions:

- **Classifying Feature-Model Analyses** To give an overview on the complexity of feature-model analyses, we collect various analyses from the literature. For each identified analysis, we assess the complexity regarding the underlying computational problem (e.g., SAT or #SAT) and the number of solver queries potentially required.
- **Identifying Knowledge-Compilation Target Languages** To showcase the potential of knowledge compilation, we perform a literature survey to identify knowledge-compilation target languages and respective knowledge compilers. Further, we present a mapping between feature-model analyses and knowledge-compilation target languages, indicating which analyses are tractable (i.e., solvable in polynomial time) for which target language.
- **Evaluating Compiler Scalability** Finally, we analyze the scalability of different knowledge-compilation target languages by evaluating the identified knowledge compilers on 53 industrial feature models.

The remainder of this article is structured as follows: In Section 2 and Section 3, we explain feature-model analysis using a running example and present the methodology of our literature review. In Section 4, we present a list of feature-model analyses and classify them according to the type of computational problem that needs to be solved. In Section 5, we list knowledge-compilation target languages that enable polynomial-time queries for common feature-model analyses. In Section 6, we empirically evaluate the scalability

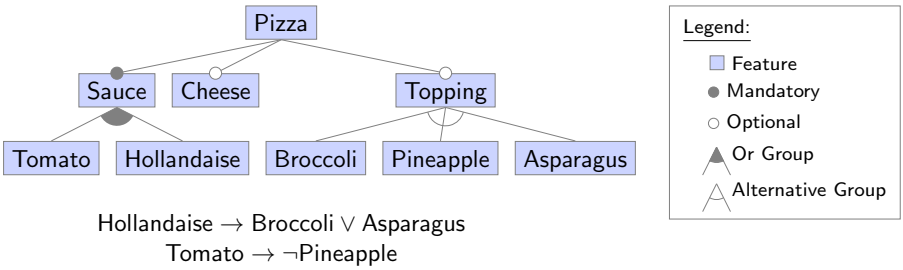

Fig. 1 Running Example: Feature Model Representing Pizza

of compiling industrial feature models to different knowledge-compilation artifacts with publicly available knowledge compilers. Finally, we review related work and conclude the paper in Section 7 and Section 8.

2 Background & Running Example

A *product line* describes a family of products that share certain characteristics, called *features* [1–5]. Each product corresponds to a composition of features (a *configuration*) that appear in the product line. Typically, not every configuration leads to a useful product (i.e., it may contain errors or may be inefficient to produce). For instance, a car might be limited to have one type of gearbox, either automatic or manual.

Feature models are the de facto standard for specifying the valid configurations of a product line [2, 10, 11]. A feature model consists of a hierarchy of features (the *feature tree*) and additional propositional cross-tree constraints that further limit the set of valid configurations [11, 12]. In Figure 1, we show an example feature model that describes the valid configurations of a pizza product line. Each pizza requires a sauce indicated by the MANDATORY flag. As indicated by the OR group, a customer can select tomato or hollandaise as sauce (or both). A pizza may further include cheese as shown by the OPTIONAL flag. Finally, exactly one of three toppings (i.e., broccoli, pineapple, or asparagus) can be selected as part of an ALTERNATIVE group. In addition to the described feature tree, two cross-tree constraints below the feature tree, expressed as propositional formulas, specify further limitations: A customer cannot select the pineapple topping together with tomato sauce and needs to select either a broccoli or asparagus topping when selecting hollandaise sauce.

Formally, we define a feature model as a tuple $FM = (F, \Phi)$ over a set of features F and a set of constraints Φ, which includes both hierarchical and cross-tree constraints. A *configuration* $C = (I, E)$ consists of two sets $I, E \subseteq F$ containing included and excluded features, respectively. A feature cannot be included and excluded in the same configuration (i.e., $I \cap E = \emptyset$). A configuration that satisfies all constraints Φ imposed by the feature model is called *valid*. If all features are either selected or deselected (i.e., $I \cup E = F$), the configuration is called *complete*. Otherwise, it is called *partial* [2].

Table 1 Translation of Feature Models into Propositional Formulas (**R**oot, **C**hild, **P**arent)

Modeling Construct	Propositional Formula	Intuition
Root feature	R	R is always required
Optional feature	$C \Rightarrow P$	C requires P
Mandatory feature	$C \Leftrightarrow P$	Optional + P requires C
Or group	$\bigvee_{1 \leq i \leq n} C_i \Leftrightarrow P$	Optional + P requires at least one C_i
Alternative group	$\bigvee_{1 \leq i \leq n} C_i \Leftrightarrow P \wedge$ $\bigwedge_{1 \leq i < j \leq n} \neg(C_i \wedge C_j)$	Optional + P requires exactly one C_i

To employ standardized solvers for automated analyses, feature models are commonly translated to propositional logic [2, 7, 12, 37–39]. Every hierarchical constraint can be translated to semantically equivalent propositional logic [2, 11] by applying translation rules. Table 1 shows translations that are commonly employed in feature-model analysis [2, 19, 37, 38]. Note that multiple semantically equivalent translations for such constraints have been considered in the literature [2, 40]. In addition, the cross-tree constraints below the feature tree are typically expressed as propositional formulas [41]. Thus, without a loss of generality, we consider Φ to be a conjunction (as every constraint needs to be satisfied) of propositional formulas. The set of satisfying assignments of Φ is equivalent to the set of valid configurations for the feature model *FM*. As the analysis of satisfying configurations is known to be computationally complex, highly-optimized logic-based tools (i.e., *solvers*) can be invoked on Φ to reason about the configuration space described by the feature model *FM* [7, 9, 11, 12, 38].

Two well-known solver classes for feature-model reasoning are SAT [42, 43] and #SAT [44–46] solvers. A SAT solver is a tool that, given a propositional formula ϕ, determines whether ϕ is satisfiable or not. That is, $\text{SAT}(\phi) \equiv \text{true}$ when ϕ has at least one satisfying assignment and $\text{SAT}(\phi) \equiv \text{false}$ otherwise (i.e. ϕ contains some contradiction). A #SAT solver also takes as input a propositional formula ϕ, but determines *how many* satisfying assignments it has, thus generalizing the functionality of a SAT solver. That is, $\#\text{SAT}(\phi) \equiv n$ when ϕ has exactly n satisfying assignments. In Section 4, we give a detailed account of how SAT and #SAT solvers can be applied to analyze the configuration spaces described by a feature model.

CNF, a specialization of *NNF*, is a common representation for propositional formulas. A formula is in *negation normal form (NNF)* when all its negations occur only in literals (so, conjunctions and disjunctions cannot be negated). Further, a formula is in *conjunctive normal form (CNF)* when it is a conjunction of clauses (i.e., disjunctions of literals) [47]. CNF in particular is highly relevant because both SAT and #SAT solvers typically require it as input format.

To solve arbitrary SAT and #SAT problems, exponential time is required in the worst case, as SAT is NP-complete [48] and #SAT is #P-complete [49]. The complexity of these problems can be a particular issue if numerous similar solver queries should be executed. A *query* is a computational operation

on a given instance (e.g., a feature model) without adapting the instance [25]. To improve query performance, *knowledge-compilation target languages*, which typically enable faster querying than CNF, can be used. With *knowledge compilation*, a formula is compiled once into a *knowledge compilation artifact* (i.e., an instance of a knowledge-compilation target language), which can then be reused for numerous *tractable* (i.e., solvable within polynomial time complexity) queries, in the hope of reducing the overall required query time. In Section 5, we review knowledge-compilation target languages that are known to improve the asymptotic runtime of feature-model analyses (i.e., making them tractable).

3 Methodology

Our overall goal is to elaborate on the potential of applying knowledge-compilation to feature-model analysis. Thus, we aim to provide a reasonable representation of problem settings in feature-model analyses and the applicability of knowledge-compilation target languages for those problem settings. To this end, we performed (1) an expert survey to gather relevant feature-model analyses, (2) a literature survey to collect knowledge-compilation target languages and knowledge compilers (3), a GitHub search to identify further compilers. In the following, we describe our methodology used to extract information on feature-model analyses (results discussed in Section 4), knowledge-compilation target languages (results discussed in Section 5), and knowledge compilers (results discussed in Section 6).

Gathering Feature-Model Analyses

To motivate the relevance of knowledge compilation for feature-model analyses, we aim to showcase algorithmic problems relevant in the feature-modeling domain. We identify relevant problems in two phases: First, we collect feature-model analyses considered in the literature and applied in practice. Second, we categorize each analysis into the type of algorithmic problem (e.g., SAT or #SAT) that needs to be solved for the analysis. Further, we provide worst-case complexities with regard to the number of computations required for each analysis.

In the first phase, we performed an expert survey to identify articles based on our expertise in the feature-modeling domain. Hereby, we consider work that proposes analyses or already existing surveys on feature-model analysis [1, 2, 26, 50]. Each of the authors suggested a variety of articles to include in the survey. While our list of considered articles may fail to cover all available feature-model analyses, we expect that our list is a reasonable representation of analyses and in particular of underlying algorithmic problems in the literature.

In the second phase, we used the list of gathered analyses to derive an initial set of problem classifications. Afterwards, we performed a first categorization of each analysis into the different problem classes. Then, we iteratively updated this categorization according to feedback of each author, also evaluating and

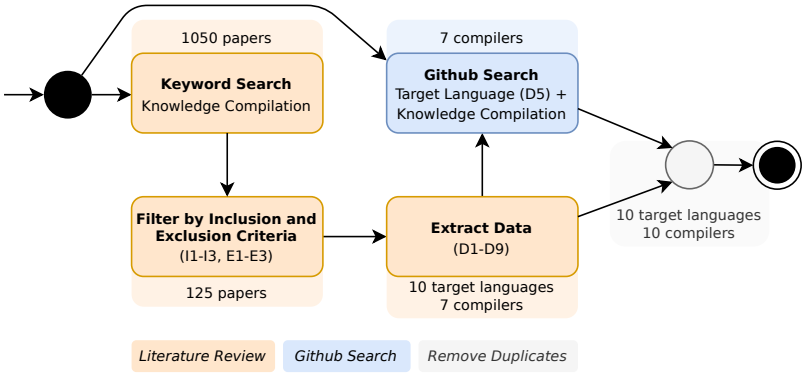

Fig. 2 Methodology for Gathering Knowledge-Compilation Target Languages & Knowledge Compilers

eventually updating the initially proposed classifications. Note that while we aim to identify popular and efficient solutions, we may not identify the least complex solution for every considered feature-model analysis.

Gathering Knowledge-Compilation Target Languages & Knowledge Compilers

To accurately assess the potential of knowledge compilation in practice, we elaborate on the capabilities of knowledge-compilation target languages and scalability of publicly available knowledge compilers. To this end, we first conduct a literature survey to identify considered knowledge-compilation target languages and published knowledge compilers. Second, we perform a GitHub search for each of the identified knowledge-compilation target languages to collect further compilers. In Figure 2, we provide an overview of the methodology for our survey.

In the first phase, depicted on the left side of Figure 2, we start by gathering relevant literature. We perform a keyword search in common databases, namely *ACM*, *Springer*, and *IEEE*. Hereby, we search for papers including the term `"knowledge compilation"`. After collecting the set of papers, we filter according to our inclusion and exclusion criteria described in Table 2 to remove work not related to knowledge compilation of translating propositional formulas to target languages enabling more tractable queries. Hereby, we also exclude knowledge compilation of higher order logics (e.g., first-order logic) as we limit our elaborations on feature models representable as propositional logic (cf. Section 2).

After reducing the considered literature set, we aim to identify considered knowledge-compilation target languages and referenced knowledge compilers. In particular, we extract the data shown in Table 3. With D1–D4, we collect meta data about the submission. For D5, we collect all knowledge-compilation target languages mentioned in the paper, which we later use for our GitHub

Table 2 Inclusion (I) and Exclusion (E) Criteria

Index	Description
I1	Knowledge compilation is employed on propositional logic.
I2	Considered languages are tractable for queries relevant for feature-model analysis.
I3	Full text of the publication is available.
E1	Publication is not in English.
E2	Publication is not peer-reviewed.
E3	Publication is subsumed by another paper in the dataset.

Table 3 Data Extraction

Index	Data	Description
D1	Title	Title of Paper
D2	Authors	Authors of Paper
D3	Venue	Publishing Venue
D4	Year	Year of Publishing
D5	Target Languages	List of Knowledge-Compilation Target Languages
D6	Compiler Availability	Availability Score: (1) N/A, (2) Not Public, (3) Public
D7	Compiler URL	URL to Compiler if Publicly Available
D8	Input Language	Input Language for the Compiler (e.g., CNF)
D9	Target Language	Target Language of the Compiler (e.g., BDD)

search. With D6, we record the availability of a compiler using the three categories (1) no compiler presented, (2) compiler is presented but not publicly available, and (3) compiler is publicly available. If available, we collect the URL to the considered compiler in D7. Lastly, we collect the input and target language of the compiler with D8 and D9, respectively.

We expect that some compilers have not been published in peer-reviewed work. Thus, after extracting the data in our literature survey, we perform a search on GitHub looking for further knowledge compilers. Hereby, we perform two searches on the GitHub database:

1. `"knowledge compilation" in:name,description,topics,readme`
2. For each target language considered in some paper of the survey (D5): `<target_language> in:name,description,topics,readme`

During the GitHub search, we include tools that accept dimacs (CNF) as input and compile to a considered target language. If the README and documentation do not provide sufficient information on how to build or run the tool, we exclude it. For BDDs, we use the knowledge compilers that we already identified in previous work with a similar procedure [21]. There, we collected hundreds of BDD libraries available on GitHub and extracted three popular tools which we use directly instead of performing another GitHub search.

By combining the set of knowledge compilers found in the survey and the GitHub search we aim to find a large variety of knowledge compilers. While we likely miss some available knowledge compilers, we expect to collect a reasonable representation of state-of-the-art knowledge compilers. With the

collected compilers, we examine their scalability for feature-model analysis in our empirical evaluation (cf. Section 6).

4 Classifying Feature-Model Analyses

Feature-model analyses depend on a variety of complex solving techniques, such as SAT [2] or #SAT [7, 9]. In practice, the majority of analyses are commonly computed by repetitively invoking a black-box solver on similar formulas [2, 10, 11, 26, 27, 29, 51–53]. In this section, we discuss various analyses to showcase the potential benefits of employing knowledge compilation. We present relevant solving techniques by classifying each analysis with regard to the algorithmic problem that it is typically reduced to. We also give an asymptotic assessment on the computational complexity of each analysis.

To assess the complexity of the gathered analyses, we elaborate on the number of queries that are required for an analysis on a given feature model. Hereby, we consider two dimensions: First, we provide the number of queries asymptotically required to compute a result for the corresponding analysis. Second, we differentiate between *static* analyses that are one-time computations on the feature model and *dynamic* analyses that can be applied numerous times with differing inputs, for instance in interactive settings. For example, while analyzing a partial configuration only requires one query, an interactive setting may require various analyses on different partial configurations but on the same unchanged feature model (dynamic). Other analyses just depend on the feature model which we consider as fixed and, thus, multiple computations do not yield additional results (static).

4.1 SAT-Based Analyses

In this section, we present different feature-model analyses that are typically based on SAT. In particular, given a propositional formula representing the feature model, possibly imposed with assumptions that (de-)select features, the satisfiability is checked to compute results for a given analysis. In the literature, a large variety of analyses have been proposed that are used for debugging, testing, sampling, and configuring [2, 27, 54–56]. Table 4 gives an overview of the collected feature-model analyses. The column *Queries* indicates the number of queries that are required for the analysis in the worst case (e.g., if implemented naively). *Repeatability* indicates whether an analysis is static or dynamic.

Consistency of Feature Models

For the first category, the analyses depend on the consistency of the entire feature model, which indicates whether at least one configuration is valid. The cross-tree constraints of a feature model may include a contradiction within each other or with tree constraints. In this case, we consider a feature model

Table 4 Analyses Based on Consistency ($|F|$ = #Features, $|VC|$ = #Valid Configurations)

Type	Analysis	Queries	Repeatability
Cons. of Feature Models	Void Feature Model [2]	1	Static
Cons. of Features	Core Features [2]	$O(\|F\|)$	Static
	Dead Features [2]	$O(\|F\|)$	Static
	Variant Features [2]	$O(\|F\|)$	Static
Cons. of Partial Conf.	False-Optional Features [2]	$O(\|F\|)$	Static
	Atomic Sets [54]	$O(\|F\|^2)$	Static
	Valid Partial Configuration	1	Dynamic
	Decision Propagation [51]	$O(\|F\|^2)$	Dynamic
	Conditionally Core Features [2]	$O(\|F\|)$	Dynamic
	Conditionally Dead Features [2]	$O(\|F\|)$	Dynamic
	t-Wise Sampling [27]	$O(\|F\|^t)$	Static
	Non-Uniform Random Sampling	$O(1)$	Dynamic
	Most-Enabled-Disabled [55]	$O(1)$	Static
	One Enabled [55]	$O(\|F\|)$	Static
	One Disabled [55]	$O(\|F\|)$	Static
Cons. of Formulas	Redundant Constraint [2]	$O(\|\Phi\|)$	Static
	Feature-Model Edits [56]	$O(1)$	Static

to be inconsistent. A consistent feature model is generally required for follow-up analyses and, thus, commonly checked first. Checking the consistency of a feature model typically corresponds to a single SAT query.

Definition 1 (Consistency of Feature Models)**.** A given feature model $FM = (F, \Phi)$ is considered consistent if and only if its constraints in Φ do not contradict each other. Determining feature model consistency can be reduced to a satisfiability check as follows: $CO(FM) \equiv SAT(\Phi)$

The consistency of a feature model is used to check for a void [2] (i.e., inconsistent) feature model, which indicates a major design error. With a void feature model, it is impossible to derive products as every possible configuration of features is invalid. For a given feature model, a *single* consistency-query is sufficient for checking voidness. The feature model depicted in our running example is consistent. For instance, the configuration {Pizza, Sauce, Tomato} violates no constraint.

Consistency of Features

For many analyses, the consistency of the entire feature model is not sufficient. Some depend on the consistency of features, which indicates whether the feature is included in at least one valid configuration [2, 38]. Checking the consistency of numerous features typically requires solving multiple SAT problems, up-to the total number of features in a feature model in the worst case.

Definition 2 (Consistency of Feature)**.** A given feature $f \in F$ is considered consistent in the feature model $FM = (F, \Phi)$ if and only if the feature f can be

selected without violating any constraints in Φ. The consistency of a feature can be reduced to a satisfiability check as follows: $CO(FM_f) \equiv SAT(\Phi \wedge f)$.

The consistency of features is, for instance, used to identify *dead features* [2] which do not appear in any valid configuration. To detect all dead features, we need to solve a SAT problem for every single feature in the worst case [11]. Analogously, detecting *core* features (i.e., features that appear in every valid configuration) and *variant* features (i.e., features that appear in some, but not all valid configurations) requires a SAT call for each feature in the worst case, too. The core, dead, and variable features of a feature model correspond to the *backbone* of its formula, which can be computed efficiently in practice [57]; however, a linear number of SAT calls is still required in the worst case. Each status of a feature, namely core, dead, and variant, is fixed for one version of a feature model. Hence, we consider each computation to be static.

In our running example, Sauce is a core feature as it is a mandatory child of the (always required) root feature. Thus, it is necessary to provide at least one type of sauce to sell pizza. Another example is Pineapple, which is a dead feature, possibly indicating a design error in the feature model.

Consistency of Partial Configurations

Various consistency-based analyses operate on partial configurations (i.e., a configuration that includes and excludes several, but not all features) [2]. A partial configuration is considered consistent if and only if there is at least one valid configuration that includes and excludes all features that are respectively included and excluded by the partial configuration. Analogously to consistency of features, computing the consistency of one partial configuration typically corresponds to a SAT call under assumptions (i.e., assigning literals a truth value). Depending on the analysis, it may be required to compute the consistency of various partial configurations.

Definition 3 (Consistency of Partial Configurations)**.** A given partial configuration $C = (I, E)$ is considered consistent in the feature model $FM = (F, \Phi)$ if and only if the selection of all included features and deselection of all excluded features does not violate any constraints in Φ. The consistency of a partial configuration can be reduced to a satisfiability check as follows: $CO(FM_C) \equiv SAT(\Phi \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$.

The consistency of a partial configuration is regularly employed for analyses in the product-configuration process [2, 58]. For instance, checking whether the selection of a user can still lead to a valid complete configuration (i.e., is a *valid partial configuration*) and, thus, a usable product is often employed to implement configuration editors. This can be extended by *decision propagation*, which computes what features must (i.e., *conditionally core*) or cannot (i.e., *conditionally dead*) be selected anymore given a partial configuration. Decision propagation requires SAT calls up to the number of open features (i.e.,

not included or excluded in the partial configuration) for each partial configuration [51]. Typically, decision propagation is performed after every decision of the user (i.e., selection or deselection of a feature) over the remaining open features, thus requiring up to $\frac{|F|^2}{2}$ SAT calls to derive a complete configuration. As multiple configurations may be configured for a given feature model, decision propagation, valid partial configuration, conditionally core features, and conditionally dead features can be employed many times and, thus, we consider them to be dynamic analyses.

Several *t-wise sampling* algorithms depend on computing the consistency of a partial configuration [27, 59–65]. Whenever a given t-wise interaction $T = \{a, b, ...\}$ is supposed to be covered by a configuration $C = (I, E)$ with $T \subseteq (I \cup E)$, the validity of this (partial) configuration must be ensured. Additionally, the validity of a t-wise interaction itself can be checked, before attempting to cover it, by interpreting it as a partial configuration. As one t-wise sample fully covers the t-wise interactions over a feature model, multiple computations for the same t are typically not beneficial. Thus, we consider t-wise sample as static analysis.

Non-uniform random sampling computes a list of configurations of a given size. To this end, each configuration is generated by finding one random satisfying assignment of the feature-model formula. An algorithm may forbid the generation of duplicated configurations, which makes the problem of finding satisfying assignments progressively more difficult. As different random samples may intentionally be derived various times, we consider the non-uniform random sampling as dynamic.

Other sampling algorithms, such as *most-enabled-disabled*, *one-enabled*, and *one-disabled*, compute a fixed-size list of valid configurations with specific properties. Most-enabled-disabled computes exactly two configurations [55]. One with a minimal number of selected features and one with a maximal number of selected features. One-enabled computes one configuration per feature $f \in F$, which has the feature f selected and as many other features as possible deselected [55]. In contrast, one-disabled also creates one configuration per feature $f \in F$, but deselects the feature f and tries to included as many other features as possible [55]. We consider each of the three sampling approaches to be static as the respective samples are supposed to be stable for a given fixed feature model.

An *atomic set* describes features that behave as a unit (i.e., are always simultaneously included or excluded) [30]. In the worst case, the consistency of partial configurations corresponding to all pairs of features needs to be covered, leading to up to four SAT calls (i.e., $\Phi \wedge a \wedge b$, $\Phi \wedge a \wedge \neg b$, $\Phi \wedge \neg a \wedge b$, $\Phi \wedge \neg a \wedge \neg b$) for each of the pairs. To reduce this effort, other analysis results can be leveraged when computing atomic sets (e.g., the formula's backbone always corresponds to one atomic set). Computing atomic sets is static, as they remain equal for one version of a feature model.

Consistency of Formulas

Some analyses depend on arbitrary adaptations of the feature-model formula. While some analyses could be reduced to the consistency of features or partial configurations, applying assumptions to the original formula is often expressive enough.

Definition 4 (Consistency of Formula)**.** For a given feature model $FM = (F, \Phi)$, the formula Φ' corresponds to an arbitrarily adapted formula $\Phi' = (\Phi \setminus \Phi_{removed}) \cup \Phi_{added}$ with Φ_{added} and $\Phi_{removed}$ being the set of added and removed clauses. The consistency of the formula Φ' can be reduced to a satisfiability check as follows: $CO(\Phi') \equiv SAT(\Phi')$.

When developing a feature model, some constraints may have no impact on the configuration space as their induced limitations are already expressed by other constraints [2]. Detecting such *redundant constraints* allows developers to clean their feature model to improve readability. Checking for one redundant constraint R can be performed by having a SAT call on $\Phi \setminus R \Rightarrow R$. To check all constraints for redundancy, up to $|\Phi|$ queries are required. As the constraints for a fixed feature model remain equal, we consider the analysis to be static.

There are many potentially insightful properties for comparing two feature models, such as differences in the set of features or set of cross-tree constraints [66]. While many of those depend on syntactical properties and can be easily extracted, other approaches allow insights on the semantic differences of two feature models [56, 66]. Thüm et al. [56] propose to classify edits to feature models into four types, namely generalization (i.e., added valid configurations), specialization (i.e., removed valid configurations), arbitrary edit (i.e., both added and removed), and refactoring (i.e., equal configuration space). To detect the edit type, $FM_1 \Rightarrow FM_2$ and $FM_2 \Rightarrow FM_1$ is checked.

4.2 #SAT-Based Analyses

In this section, we present analyses that are typically based on computing the number of valid configurations of a feature model. Computing the number of valid configurations enables a variety of analyses that can be used for debugging, development support, economical estimations, and statistical inferences [26]. A common method to compute the number of valid configurations is reducing the problem to #SAT [7, 9, 26, 50]. Table 5 gives an overview on the collected analyses based on #SAT indicating the complexity of each analysis w.r.t. the number of queries and repeatability.

Cardinality of Feature Models

Analogous to SAT-based analyses, the first type of #SAT-based analyses depends on analyzing the entire feature model [26]. The cardinality of feature models corresponds to the number of all valid configurations induced by the

Table 5 Analyses Based on Cardinality

Type	Analysis	Number of Queries	Repeatability
Card. of Feature Models	Variability Factor [13]	1	Static
	Variability Reduction [26]	1	Static
	Degree of Orthogonality [67]	1	Static
	Maintainability Prediction [68]	1	Static
	Configuration Relevance [26]	1	Static
	Cost Savings of Product Line [69]	1	Static
Card. of Features	Homogeneity [28]	$O(\|F\|)$	Static
	Atomic Set Candidates [26]	$O(\|F\|)$	Static
	Feature Prioritization [26]	$O(\|F\|)$	Static
	CTC Restrictiveness [26]	$O(\|F\|)$	Static
	Degree of Reuse [70]	$O(\|F\|)$	Static
	Optimize Configuring [71, 72]	$O(\|F\|)$	Static
	Payoff Threshold [50]	$O(\|F\|)$	Static
Card. of Partial Conf.	Interactive Configuration [26]	$O(\|F^2\|)$	Dynamic
	Uniform Random Sampling [16]	$O(\|F\|)$	Dynamic
	Rating Feature Interactions [26]	$O(\|F^2\|)$	Dynamic
Card. of Formula	Rating Errors [9]	1	Dynamic
	Variability of Feature Subsets [26]	1	Dynamic

feature model. The cardinality of a feature model can be computed by invoking a #SAT solver on the formula representing the feature model.

Definition 5 (Cardinality of Feature Models)**.** The cardinality $\#FM$ of a feature model $FM = (F, \Phi)$ is defined as the cardinality of the set of valid configurations $|VC|$. The feature-model cardinality can be reduced to computing a #SAT problem as follows: $\#FM \equiv \#\text{SAT}(\Phi)$.

Various analyses depend on the cardinality of a feature model [26]. The *variability factor* describes the share of valid configurations compared to all $2^{|F|}$ configurations [13]. *Variability reduction* corresponds to using the change in cardinality after a new revision of the feature model to, for instance, detect errors [26]. Still, all analyses depend on a single query per feature model [26].

In our running example in Figure 1, there are 14 valid configurations (i.e., 14 different pizzas that can be ordered). As there are overall $2^9 = 512$ theoretically possible configurations, the variability factor is $\frac{14}{512} \approx 2.73\%$.

Cardinality of Features

Other analyses depend on knowing the number of valid configurations that contain a certain feature (i.e., the cardinality of that feature). Computing the cardinality of a feature is typically reduced to invoking a #SAT solver on the formula representing the feature model with addition of an assumption for that feature.

Definition 6 (Cardinality of Features)**.** The cardinality $\#FM_f$ of a feature $f \in F$ given a feature model $FM = (F, \Phi)$ is defined as the cardinality of the

set of valid configurations that contain the feature f (i.e., $|\{C = (I, E) \mid C \in VC, f \in I\}|$). The feature cardinality can be reduced to computing a #SAT problem as follows: $\#FM_f \equiv \#\text{SAT}(\Phi \wedge f)$.

The cardinality of features can be used for economical estimations when considering to develop a family of products as product line [50, 70]. For instance, it may be beneficial to *prioritize features* that appear in many valid configurations to allow building more distinct products earlier in the development process [50]. Further, one may consider the *payoff threshold*, which indicates when the additional effort for building a feature for reuse in a product line is worth. A cardinality of the feature being higher than the payoff threshold may indicate that reusing the feature is beneficial. For both analyses, namely *feature prioritization* and *payoff threshold*, the cardinality for each feature of interest needs to be computed inducing linear effort in the worst case (i.e., $O(|F|)$).

Cardinality of features can also be employed to extract information about the variability of the feature model [26]. *Homogeneity* describes the similarity of valid configurations induced by a feature model [28]. The *degree of reuse* indicates the benefits of developing a software product line compared to standalone products by comparing the number of valid configurations induced by common features (i.e., features appearing in more than two valid configurations) and all features [70]. All features in an atomic set share the same cardinality. Hence, the cardinality of features can be used to identify *atomic set candidates* to reduce the number of interactions that need to be checked [26]. All three metrics listed above depend on the share of valid configurations each feature appears in and, thus, requires by definition the cardinality of each feature. *Cross-tree constraint (CTC) restrictiveness* describes how the cross-tree constraints impact the share of valid configurations a feature appears in compared to the limitations of the tree hierarchy [26]. As analyzing the cross-tree constraint restrictiveness may be beneficial for every feature, up to $|F|$ invocations may be employed.

Several strategies have been proposed for ordering features to select when configuring products [71, 72]. According to Mazo et al. [72], it may be reasonable to configure features with a high cardinality first as they tend to be more important in the product line. Selecting features with a small cardinality first may also be useful to accelerate the configuration process, as selecting these feature vastly reduces the number of remaining valid configuration and, thus, potentially the number of remaining decisions [71]. Fully sorting the features by their cardinality requires up to $|F|$ #SAT calls. Given a fix feature model revision, all analyses dependent on the cardinality of features are static as the cardinalities remain equal.

Cardinality of Partial Configurations

Further analyses are based on the number of valid configurations that include some and exclude other features (i.e., conform to a partial configuration) which

we refer to as *cardinality of a partial configuration*. Note that the cardinality of partial configurations is a generalization of the cardinality of a features.

Definition 7 (Cardinality of Partial Configurations)**.** The cardinality $\#FM_{C'}$ of a partial configuration $C' = (I', E')$ given a feature model $FM = (F, \Phi)$ is defined as the cardinality of the set of valid configurations that include all selected features and exclude all deselected features f (i.e., $\#FM_{C'} = |\{C = (I, E) \mid C \in VC, I' \subseteq I, E' \subseteq E\}|$). The cardinality of a configuration can be reduced to computing a #SAT problem as follows: $\#FM_{C'} \equiv \#\text{SAT}(\Phi \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e)$

The cardinality of partial configurations can be used for *uniform random sampling*, which computes a sample of configurations where each valid configuration has the same chance to appear in the sample [16, 26, 52]. While several algorithms have been proposed [22, 73], uniform random sampling can be reduced to #SAT [52]. Features are added to the configuration iteratively and for each feature the probability with which a feature needs to be included is computed. The probability is derived from the relative share of valid configurations the feature appears in based on the current partial configuration (i.e., the already included and excluded features). In the worst case, the probability for each feature (i.e., $O(|F|)$) needs to be computed requiring one #SAT call per feature. Depending on the use case, many configurations for a single feature model may be required and, thus, we consider the analysis to be dynamic.

While t-wise sampling is generally based on SAT (cf. Section 4.1), the cardinality of partial configurations can be used to potentially improve the sampling process. The runtime and sample size depends on the *prioritized feature interactions* for coverage. Explicitly prioritizing interactions that appear in few valid configurations may be beneficial as interactions with a high cardinality are more likely to be covered anyhow [26]. As interactions remain equal for a fixed feature model, we consider the analysis to be static.

When *interactively configuring* a product, it may be beneficial to indicate the consequences of a decision for the user. For instance, providing the number of remaining valid configurations after each decision may be helpful for the user [26]. This information can be extracted by computing the cardinality of partial configurations for each open (i.e., neither selected nor deselected) feature. Analogously to decision propagation, the computation needs to be done for all remaining features for each selection made and, thus, requires $O(|F|^2)$ #SAT calls. Further, many configurations can be derived, making the analysis dynamic.

Cardinality of Formulas

Analogously to SAT-based analyses, assumptions are not necessarily sufficient for #SAT-based analyses. Model counting on adapted formulas (e.g., by adding constraints or applying a projection) allows for further analyses.

Table 6 Analyses Based on AllSAT

Type	Analysis	Number of Queries	Repeatability
Model Enumeration	All Products [2]	1	Static
	Filter [2]	1	Dynamic

Definition 8 (Cardinality of Formulas)**.** For a given feature model $FM = (F, \Phi)$, the cardinality $\#\Phi'$ of a formula corresponds to the number of valid configurations induced by an arbitrarily adapted formula $\Phi' = (\Phi \setminus \Phi_{removed}) \cup \Phi_{added}$ with Φ_{added} and $\Phi_{removed}$ being the set of added and removed clauses. The cardinality of the formula Φ' can be reduced to a #SAT problem as follows: $\#\Phi' = \#\mathrm{SAT}(\Phi')$.

The cardinality can be used to *rate errors* [9]. For instance, an error may appear if one of two features A and B is present. Adding a constraint $A \vee B$ to the formula and then invoking a #SAT solver results in the number of valid configurations the error appears in. An error that appears in many valid configurations may be more critical than another error that has a similar severity but appears in fewer valid configurations. Rating an error requires one #SAT call. However, for the same system there may be many errors and, thus, we consider the analysis to be dynamic.

In some cases, one is only interested in the *variability of a subset of features* [9]. For instance, only considering safety-critical features may be beneficial. Computing the variability for a feature subset can be done by performing a projection [74] (also called slice in the feature-modeling domain [75, 76]) and then invoking #SAT once. We further discuss projection in Section 4.4. Alternatively, projected model counting can be applied [74]. Note that projected model counting is considered to have a higher computational complexity than regular model counting [74, 77]. In particular, projected model counting is in $\#\mathrm{P}^{\mathrm{NP}}$ while model counting is #P-complete [74]. As potentially numerous feature subsets can be analyzed, we consider the analysis to be dynamic.

4.3 AllSAT-Based Analyses

For the third category, we present two analyses based on enumerating valid configurations induced by a feature model. Table 6 provides an overview on the analyses based on enumeration.

In some cases it may be beneficial to compute *all products* that can be derived from feature model [2]. The set of products (or valid configurations) could then be used to identify modeling errors or as test cases. As the set of valid configurations is equivalent to the set of satisfying assignments for the corresponding formula, computing all products can be reduced to solving one AllSAT problem. Often one is not interested in all valid configurations but a subset. A *filter* corresponds to deriving only valid configurations that conform to a partial configuration [2]. Computing all valid configurations for one filter

Table 7 Analyses Based on Algebraic Operations

Type	Analysis	Number of Queries	Repeatability
Algebraic	Feature-Model Slicing [76]	1	Dynamic
	Feature-Model Composition [78]	1	Dynamic
	Feature-Model Differences [66]	1	Dynamic

corresponds to a single AllSAT call. As there are potentially numerous filters, the analysis is dynamic.

4.4 Algebraic-Based Analyses

Some feature-model analyses are also based on algebraic operations, such as building the conjunction of two feature models. Table 7 shows an overview on the feature-model analyses based on algebraic operations.

Feature-model slicing removes a set of features from a feature model without changing the dependencies between other features [75, 76]. Given a feature model $FM = (F, \Phi)$ and a feature set $F_S \subseteq F$ the slice of a feature model corresponds to $FM' = (F \setminus F_S, \Phi')$ such that the set of all satisfying assignments of Φ' is given by $A_{\Phi'} = \{a \setminus F_S \mid a \in A_\Phi\}$ where A_Φ is the set of satisfying assignments of Φ. Slicing a feature entails creating new clauses within the feature-model formula, which have to be checked for redundancy. The amount of new clauses is dependent on which feature is being sliced. In other domains, such a transformation for propositional formulas is also known as *existential quantification* [75], *projection* [74], and *forgetting* [25].

Feature-model composition corresponds to building a feature model representing the union of set valid configurations induced by two other feature models. The composition can be computed by a conjunction as follows: Given two feature models $FM_1 = (F_1, \Phi_1)$ and $FM_2 = (F_2, \Phi_2)$ the composition corresponds to $\Phi_1 \wedge \Phi_2$. While computing the composition for two feature models always computes equivalent results, a given feature can be compared to numerous feature models. Hence, we consider the analysis to be dynamic.

Feature-model differences compare two feature models with respect to the underlying formula Φ [66]. Acher et al. [66] compute a new feature model representing the differences in configuration spaces of the two considered feature models. To compute this feature model, Acher et al. [66] derive a formula which is satisfied by each assignment that satisfies exactly one of the corresponding feature models. Computing the feature-model differences depends on conjunction and negation of feature-model formulas [66]. For two feature models $FM_1 = (F_1, \Phi_1)$ and $FM_2 = (F_2, \Phi_2)$, the formula $(\Phi_1 \wedge \neg\Phi_2) \wedge (\Phi_2 \wedge \neg\neg\Phi_1)$ represents the valid configurations that appear in exactly one of the feature models. Analogously to feature-model composition, we consider the analysis to be dynamic as a fixed feature model can be compared to various feature models.

4.5 Summary

The potentially high number of queries on the same feature model motivates employing knowledge compilation. All 40 analyses presented above depend on computational complex problems (i.e., NP-complete or harder). 32 of the analyses potentially depend on multiple queries on the same feature model. Further, several analyses are often invoked multiple times which results in even more queries on the same formula.

Currently, it is not trivial for researchers and practitioners to select a promising knowledge-compilation target language. As the presented feature-model analyses are based on different computational problems (e.g., SAT or #SAT), the queries of interest vary depending on the use case. Thus, identifying a knowledge-compilation target language that both (1) supports fast queries of interest and (2) can be compiled efficiently, requires expertise on capabilities and performance of target languages.

5 Identifying Knowledge-Compilation Target Languages

In this section, we present all knowledge-compilation target languages we identified in our survey that are tractable for at least one feature-model analysis. A computational problem is *tractable* for a given knowledge-compilation target language if and only if an artifact of that language allows computing a result for the given problem in polynomial time with respect to the size of the artifact [25]. Further, we consider a feature-model analysis to be tractable for a target language if its underlying computational problem that needs to be solved is tractable for that target language. We envision that this knowledge on tractable analyses can guide researchers and practitioners to select suitable knowledge-compilation target languages for implementing reasoning engines. In the following, we describe defining properties for each target language, provide simple examples, and list the types of problems that are tractable when solved by using a knowledge-compilation artifact (i.e. an instance of the respective target language).

5.1 Survey Results

In our literature survey (cf. Section 3), we processed a total of 1,050 publications. After applying inclusion and exclusion criteria, 125 papers remained that address knowledge compilation of propositional formulas to knowledge-compilation target languages allowing tractable queries relevant for feature-model analysis.

Overall, we identified 10 target languages with overall 22 variants relevant for feature-model analysis (considering tractable queries). In Table 8, we show the target languages we identified, their variants, and references that discuss these target languages in further detail. In the following section, we limit the elaborations on the main target languages we identified and refer to the listed

Table 8 Literature Survey Result: Identified Knowledge-Compilation Target Languages

Target Language	Variants	References
Horn	Horn	[80]
EPCCL	EPCCL	[81]
DNF	DNF	[82]
DNNF	DNNF	[83, 84]
d-DNNF	d-DNNF	[85–87]
	Decision-DNNF	[85–87]
	s-d-DNNF	[84, 85, 87]
	s-Decision-DNNF	[84, 85, 87]
EADT	EADT	[88]
SDD	SDD	[35, 89]
	ZSDD	[90]
	Decision-SDD	[91]
BDD	BDD	[92]
	FBDD	[93]
	OBDD	[92]
	ROBDD	[92]
	ZBDD/ZDD	[94]
	C-OBDD	[95]
	C-NSOBDD	[95]
	DSDBDD	[96]
	TBDD	[97]
MODS	MODS	[25]

publications for details on the variants (cf. Table 8). For BDDs, we consider *reduced ordered* BDDs (ROBDDs) as default if not stated otherwise which follows the literature on BDDs [23, 79]. Note that the list of references in the rightmost column does not include every work that mentions the target language; rather, we picked publications that provide evidence for our claims and discussions in the remainder of this section.

5.2 Knowledge-Compilation Target Languages

In Table 9, we provide an overview on the tractability of each category of feature-model analysis we described in Section 4 for the identified knowledge-compilation target languages. A ✓ indicates the feature-model analysis is tractable for the respective knowledge-compilation target language. For instance, CNFs (which we include as a baseline for our comparison) are not tractable for any of the considered analyses, while MODS are tractable for all analyses. For CNF, DNF, DNNF, d-DNNF, PI, IP, and MODS the mapping to tractable feature-model analysis is based on the *knowledge-compilation map* provided by Darwiche and Marquis [25]. The remaining mappings are based on literature targeting the specific target language [80, 81, 88, 89, 92, 98]. Note that slicing (i.e., forgetting) is not tractable for OBDDs but for ROBDDs which we consider here [92, 98].

Consistency of formula and cardinality of formula cannot be generally applied for knowledge compilation. Both analysis types, depend on arbitrarily changed formulas and, thus, not on a fixed feature-model instance. While some

Table 9 Tractability of Feature-Model Analyses for Knowledge-Compilation Target Languages

	SAT-Based			#SAT-Based			AllSAT	Algebraic		
	CO	CO_f	CO_C	#*FM*	#FM_f	#FM_c	Enum.	Comp	Diff.	Slice
CNF										
Horn	✓	✓	✓							
EPCCL	✓	✓	✓							
DNF	✓	✓	✓				✓			✓
DNNF	✓	✓	✓				✓			✓
d-DNNF	✓	✓	✓	✓	✓	✓	✓			
EADT	✓	✓	✓	✓	✓	✓	✓			
SDD	✓	✓	✓	✓	✓	✓	✓			
PI	✓	✓	✓	✓	✓	✓	✓			✓
IP	✓	✓	✓	✓	✓	✓	✓			
ROBDD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MODS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

changes (e.g., adding an optional feature) may be applicable to a knowledge-compilation artifact in polynomial time, other changed formulas vastly differ from the original feature model. Hence, we decided to exclude the analyses in this section.

Succinctness is another often considered dimension for classifying knowledge-compilation target languages [25]. Informally, succinctness gives an indication on the expected size of an artifact in a given target language compared to other target languages. In combination with tractability, succinctness can be used to identify promising target languages for a given use case. The most succinct target language that is still tractable for the required analyses is often a promising option. Formally, a target language A is at least as succinct as another language B (i.e., $A \leq B$) if and only if for every artifact in A there is a semantically equivalent artifact in B that is not exponentially larger than the original [25]. A is strictly more succinct than B iff $A \leq B$ but $B \not\leq A$.

Figure 3 shows the succinctness relations between the identified target languages. The figure is adapted from Darwiche and Marquis [25] and is extended by some additional languages, namely EPCCL, EADT, and SDD. An arrow $A \rightarrow B$ indicates that A is strictly more succinct than B. For instance, d-DNNF is strictly more succinct than SDD [89]. Note that the figure is not complete as some succinctness relations are currently unknown [25]. The references between the lines specify the publications that indicate the succinctness relation. The green symbols indicate which analyses are tractable for the different formats. In Figure 3, a promising target language can be identified by the following characterization: the language supports all required queries, but its predecessors do not support (all of) them. Note that while succinctness is often a reasonable indicator, performance in practice may still vary. Hence, we also empirically evaluate publicly available compilers for different target languages in Section 6.

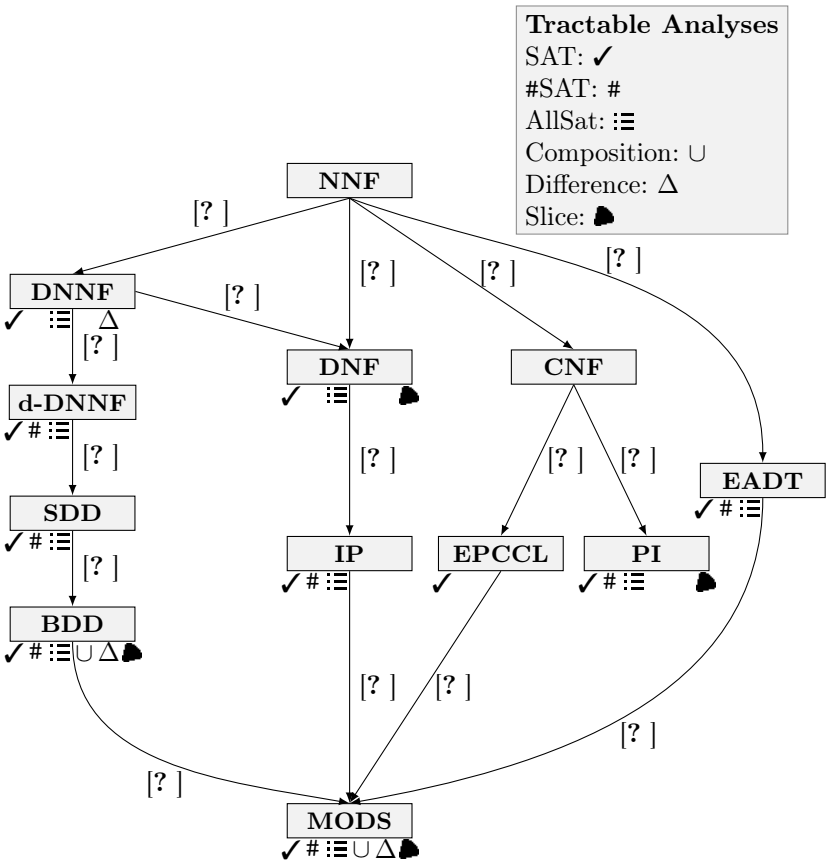

Fig. 3 Succinctness of Languages *[Any suggestions to improve this?]*

Horn Formula

A *Horn formula* is a CNF with an additional restriction, namely that each clause in a Horn formula contains at most one positive literal [80]. For instance, Equation 1 is a Horn formula, as each of the three clauses contains either one or no positive literal. Checking the satisfiability of a Horn formula has linear time complexity with respect to the literals in the formula [80]. However, Horn formulas are not complete (i.e., they cannot represent every propositional formula) [25]. As feature models typically require the full expressiveness of propositional logic [41], Horn formulas are not applicable for all feature models.

$$(A \vee \neg B \vee \neg C) \wedge (\neg B \vee \neg D) \wedge (C \vee \neg E) \tag{1}$$

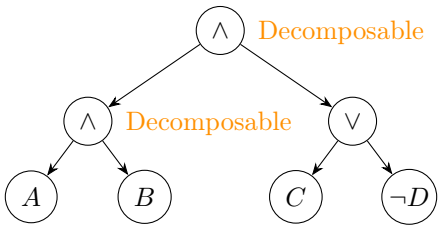

Fig. 4 Example Formula in DNNF

Each Pair Contains Complementary Literal

Each Pair Contains Complementary Literal (EPCCL) corresponds to a specialization of CNF where each pair of clauses contains complementary literals [81]. For a literal A, the literal $\neg A$ is complementary. Equation 2 shows an example formula in EPCCL. Each pair of clauses contains complementary literals: $\{A, \neg A\}$, $\{B, \neg B\}$, and $\{\neg A, A\}$, respectively. EPCCL supports polynomial time queries for consistency of features models, features, and partial configurations [81].

$$(A \vee B \vee C) \wedge (\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C) \tag{2}$$

Disjunctive Normal Form

A *disjunctive normal form (DNF)* is a disjunction of conjunctions where each of the conjunctions only consists of literals f or $\neg f$ [82]. For instance, Equation 3 is in DNF. A DNF allows polytime queries for analyses based on consistency checking, model enumeration, and slicing [25].

$$(A \wedge \neg B) \vee (B \wedge C) \vee (A \wedge C \wedge D) \tag{3}$$

Decomposable Negation Normal Form

A *decomposable negation normal form (DNNF)* is a specialization of an NNF in which each conjunction is decomposable [25]. A conjunction is *decomposable* if and only if the conjuncts share no common variables. Figure 4 shows a directed acyclic graph representing the DNNF $(A \wedge B) \wedge (C \vee \neg C)$. Both conjunctions are decomposable as they share no variables (i.e., $\{A\} \cap \{B\} = \emptyset$ and $\{A, B\} \cap \{C\} = \emptyset$). Consistency checking, model enumeration, and slicing are tractable for DNNFs [25].

Deterministic Decomposable Negation Normal Form

A *deterministic decomposable negation normal form (d-DNNF)* is a specialized NNF in which each disjunction is deterministic and every conjunction is decomposable [24, 25]. A disjunction is *deterministic* if and only if the disjuncts

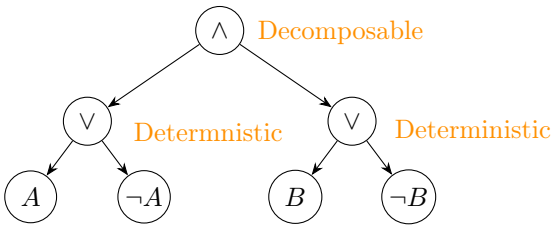

Fig. 5 Example Formula in d-DNNF

share no common solutions. Figure 5 shows a simple d-DNNF representing the formula $(A \vee \neg A) \wedge (B \vee \neg B)$ as a directed acyclic graph. The children of the conjunction ($\wedge$) share no variables (i.e., it is decomposable) and the children of each disjunction ($\vee$) share no solutions (i.e. they are deterministic). The combination of determinism and decomposability enables tractable queries for consistency checking, model counting, and enumeration [25].

Decision-DNNF, which is a subset of d-DNNF, are often applied [86, 87, 99]. Hereby, each disjunction is a decision node on a variable a with the structure $(\phi \wedge a) \vee (\tau \wedge \neg a)$ where ϕ and τ are subformulas that also conform to the properties of decision-DNNF. Due to its structure, each decision node is a deterministic disjunction by definition. Thus, all queries tractable for d-DNNF are tractable for decision-DNNF.

Smoothness is also often considered for d-DNNFs and decision-DNNFs [24, 25, 87]. A disjunction is *smooth* if all disjuncts of a disjunction share the same set of variables. While smoothness simplifies some queries, adding it does not enable more tractable queries for d-DNNF or decision-DNNF.

Extended Affine Decision Tree

An affine clause is a XOR (i.e., $\oplus$) over a set of literals. An *extended affine decision tree (EADT)* is a finite tree where each external node is a truth value (i.e., $\top$ or $\bot$) and the internal nodes are either affine decision nodes, affine decomposable conjunction, or affine decomposable disjunctions [88]. *Affine decision nodes* consists of an affine clause and two children corresponding to satisfying and not satisfying the affine clause, respectively. Alternatively, an affine decision node follows the structure: $((a_1 \oplus \ldots \oplus a_n) \wedge \Psi_l) \vee (\neg(a_1 \oplus \ldots \oplus a_n) \wedge \Psi_r)$. A conjunction is *affine decomposable* if and only if each of its conjuncts share no variables. Analogously, a disjunction is *affine decomposable* if and only if each of its disjuncts share no variables. Figure 6 shows an example EADT representing the formula $A \vee B \vee C$. The dashed edges indicate that its root term is unsatisfied while the solid line indicates that the root term is satisfied. Consistency checking, computing cardinalities, and model enumeration are tractable for EADTs.

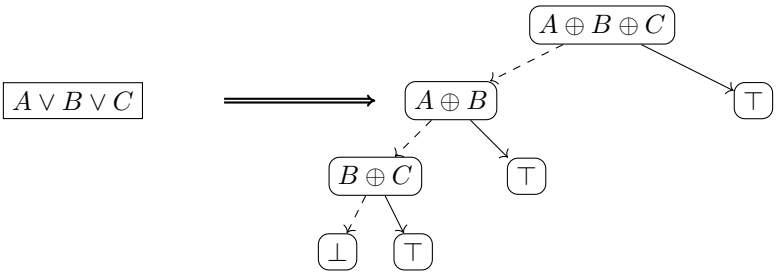

Fig. 6 Example Formula in EADT

Sentential Decision Diagram

Sentential decision diagram (SDD) is a generalization of BDD [35, 89]. Instead of having binary decision nodes on *one* variable (as in BDDs), decision nodes in SDDs may have arbitrarily many children on different disjunct sentential forms. More formally, an SDD is an (X, Y)-*partition* meaning the original formula is decomposed in the structure $(p_1(X_1) \wedge s_1(Y_1) \vee \ldots \vee (p_n(X_n) \wedge s_n(Y_n)))$ conforming additional properties. Hereby, $X_i \subseteq X$ and $Y_i \subseteq Y$ are variable sets with $X \cap Y = \emptyset$ and p_i (called *primes*) and s_i (called *subs*) are boolean functions over X_i and s_i, respectively. Such a decomposition is considered to be a (X, Y)-partition iff the primes are deterministic (i.e., share no solutions), exhaustive (i.e., $p_1 \vee \ldots \vee p_n = \top$), and satisfiable on their own (i.e., $\forall i : \mathrm{SAT}(p_i) = \top$). Figure 7 shows an example SDD representing the formula $(A \wedge B) \vee (B \wedge C)$. The (X, Y)-partition decomposes the formula in $X = \{A, B\}$ and $Y = \{C\}$. The leaves show the sub formulas separated in *prime* | *sub*. SDDs enable polytime queries for consistency checking, cardinality computations, and model enumeration.

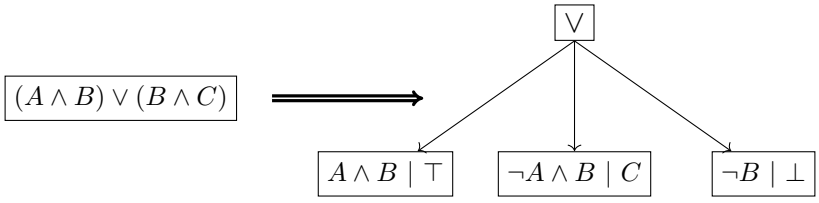

Fig. 7 Example Formula SDD

Prime Implicant

A term i (i.e., conjunction of literals) is an *implicant* of a formula F if and only if satisfying the term always satisfies F (i.e., $i \vdash F$). An implicant is *prime* if it cannot be further reduced (i.e., by removing literals) without losing the implicant property. The knowledge-compilation target language PI corresponds to a specialization of DNF where each term is a prime implicant [25]. A disjunction over all prime implicants (PI) of a formula F is also called the

Blake canonical form of F. Figure 8 shows the prime implicants and PI representation for the formula $(A \vee B) \wedge (B \vee C)$. Consistency, equivalence, and model enumeration is tractable for prime implicants [25].

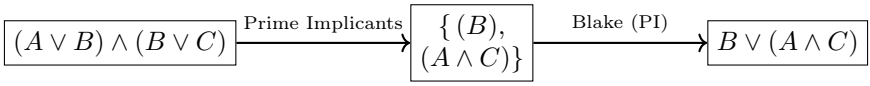

Fig. 8 Prime Implicants for Example Formula

Prime Implicate

Prime implicates are the dual problem to prime implicants: A clause i (i.e., disjunction of literals) is an *implicate* of formula F if and only if satisfying the formula always satisfies i. An implicate is *prime* if it cannot be further reduced without losing the implicate property. The knowledge compilation target language IP corresponds to a specialization of CNF where each clause is a prime implicate [25]. Figure 9 shows the prime implicates and IP representation for the formula $(A \vee B) \wedge (B \vee C)$. Consistency, equivalence, and model enumeration is tractable for prime implicates [25].

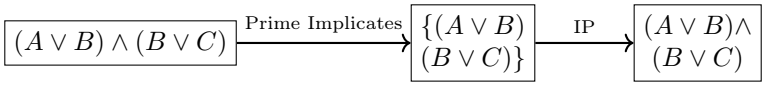

Fig. 9 Prime Implicates for Example Formula

Binary Decision Diagram

Binary Decision Diagrams (BDD) represent Boolean formulas as directed acyclic graphs with two terminal nodes $\boxed{0}$ and $\boxed{1}$. Every inner node in a BDD is associated to a variable of the formula and has precisely two outgoing edges (called high and low edge), denoting the assignment of true or false to this variable, respectively. Typically, one assumed BDDs to be *ordered* and *reduced* [92]. A BDD is ordered, when the sequence of variables on every path from the root node to the terminal nodes is the same. BDDs are reduced, when they neither contain a node with its outgoing edges incident with the same node (i.e., the assignment of the associated variable has no consequences on the outcome) nor two nodes associated to the same variable that have the same nodes incident to their high and low edges [92]. Figure 10 shows a BDD representing the formula $(A \wedge B) \vee (B \wedge C)$.

A plethora of feature-model analyses can be tractably performed on BDDs, namely consistency checking, cardinality computations, model enumerations, and some algebraic operations (i.e. diffing and composition). While forgetting variables is not tractable [25], existential quantification is tractable and could potentially be applied for slicing [92, 98].

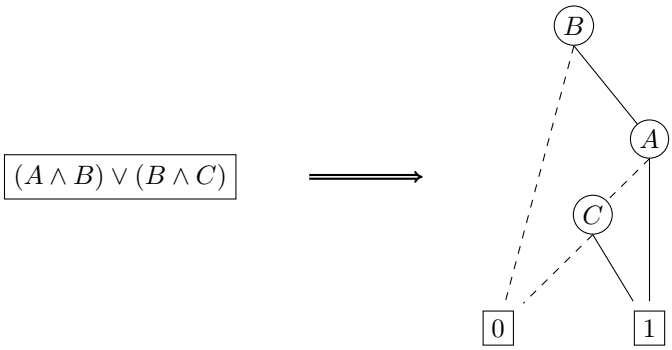

Fig. 10 Example Formula BDD

Models

The class of formulas MODS (short for *models*) is the subset of formulas in DNF that satisfy determinism and smoothness (i.e. a kind of smooth d-DDNF) [25]. In essence, MODS enumerates all satisfying assignments, as each disjunct describes one satisfying assignment. Equation 4 shows a propositional formula conforming to the restrictions of MODS. The formula is in DNF, the disjuncts share no solutions (deterministic), and contain the same variables (smooth). Each considered feature-model analysis is tractable given a MODS formula [25].

$$(A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C) \tag{4}$$

6 Evaluating Compiler Scalability

In addition to tractable queries, an important aspect for the applicability of knowledge compilation is the scalability of compiling to target languages. In our empirical evaluation, we examine the runtimes of publicly available knowledge compilers on compiling industrial feature models. The framework and input data used for the empirical evaluation is publicly available.[1]

6.1 Research Questions

RQ1 How do knowledge compilers for different target languages perform on the task of compiling industrial feature models?

The benefits of knowledge compilation (i.e., fast online queries), are only applicable for feature-model analysis if the compilers scale to industrial feature models. To examine the applicability, we first inspect the general scalability regarding compilation time for different knowledge-compilation target languages. Second, we compare publicly available compilers to give recommendations on the best-performing tools for each target language.

[1] https://github.com/SoftVarE-Group/kc-for-fmanalysis-evaluation/

Table 10 Subject Systems

Domain	Models	Features	Clauses	Cardinality
Operating System	37	94–62,482	190–343,944	10^{10}–10^{417}†
Other Software*	13	11-31,012	1–102,705	10^{3}–10^{5}†
Automotive	2	384–18,616	1,020–350,221	10^{4}–10^{1534}
Financial Services	1	771	7,241	10^{13}

* Including archivers (2), database (2), video (2), network (1), garbage collection (1), image (1), compiler (1), solving (1), web development (1), and team management (1)

† Cardinalities are incomplete due to every solver hitting timeouts for some models

RQ2 Which knowledge-compilation target languages should be used for the different types of feature-model analyses?

Typically, there is a tradeoff between tractability of queries and the complexity of compiling for selecting target languages. For each considered feature-model analysis, we aim to identify the most scalable target language that still allows polynomial queries for the respective analyses.

6.2 Subject Systems

The applicability of a knowledge-compilation artifact is highly dependent on whether real-world instances can be compiled within a reasonable amount of time. Thus, we consider industrial feature models from a variety of domains that have been considered in other empirical evaluations.

Table 10 provides an overview of the considered systems sorted into application domains. First, we consider 39 feature models from the software systems and application domain provided by Oh et al. [16]. Second, we include three CDL[2], eight KConfig[3], and one automotive feature model published by Knüppel et al. [41]. Third, our dataset contains a BusyBox[4] feature model, which was extracted by Pett et al. [100]. Fourth, we consider a feature model from the financial service domain [101, 102]. Fifth, two additional models are available as FeatureIDE [19] examples.

6.3 Knowledge Compilers

We include every found knowledge compiler that takes CNF (`dimacs`) as input and compiles to one of the knowledge-compilation target languages we identified. When collecting compilers, we identified seven compilers from the literature survey and seven compilers with the GitHub search resulting in overall ten knowledge-compilers.

Table 11 gives an overview of the identified knowledge compilers indicating the corresponding target language, the corresponding work, and a link to source files or binary. For each of d-DNNF, SDD, and BDD, our evaluation includes three compilers. We evaluate the BDD libraries `BuDDy` and `Cudd` by using the `dduerum` wrapper, we provided in previous work [21]. Further,

[2] http://ecos.sourceware.org/ecos/docs-latest/cdl-guide/language.html
[3] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html
[4] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling

we found one EADT compiler. We were not able to find compilers for other formats that accept CNF as input.

Table 11 Evaluated Knowledge Compilers

Name	Artifact	Reference	Link
`c2d`	d-DNNF	[24]	[5]
`dSharp`	d-DNNF	[87]	[6]
`d4`	d-DNNF	[86]	[7]
`MiniC2D`	SDD	[103]	[8]
`SDD`	SDD	[35]	[9]
`ZSDD`	ZSDD	[90]	[10]
`Cudd`	BDD	[104]	[11]
`BuDDy`	BDD	[105]	[12]
`CNF2OBDD`	BDD	[106]	[13]
`CNF2EADT`	EADT	[88]	[14]

6.4 Experiment Design

We invoke each listed knowledge compiler on every considered feature model. Thereby, we provide each feature model as CNF in `dimacs` format and perform ten repetitions. If a feature model was not already given as CNF, we translated it to `dimacs` using FeatureIDE v3.8.3, which uses a translation based on equivalence transformations [14]. We set a timeout of 10 minutes for each knowledge compiler, as feature models are typically analyzed either interactively or in a CI/CD environment [19, 51, 107]. In both cases, runtimes longer than a few minutes typically disrupt the working flow. Further, increasing the timeout in preliminary experiments had no substantial impact on the results. We use the default parameters of every considered tool. For each measurement, we perform ten repetitions and collect the resulting knowledge-compilation artifact and runtime for further analysis.

Technical Setup

The empirical evaluation was run on a designated server with the following specifications:

- Operating System: Ubuntu 20.04.4 LTS

[5] http://reasoning.cs.ucla.edu/c2d/
[6] https://github.com/QuMuLab/dsharp
[7] https://github.com/crillab/d4
[8] http://reasoning.cs.ucla.edu/minic2d/
[9] http://reasoning.cs.ucla.edu/sdd/
[10] https://github.com/nsnmsak/zsdd
[11] https://github.com/vscosta/cudd
[12] http://buddy.sourceforge.net/manual/main.html
[13] http://www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html
[14] http://www.cril.univ-artois.fr/KC/eadt.html

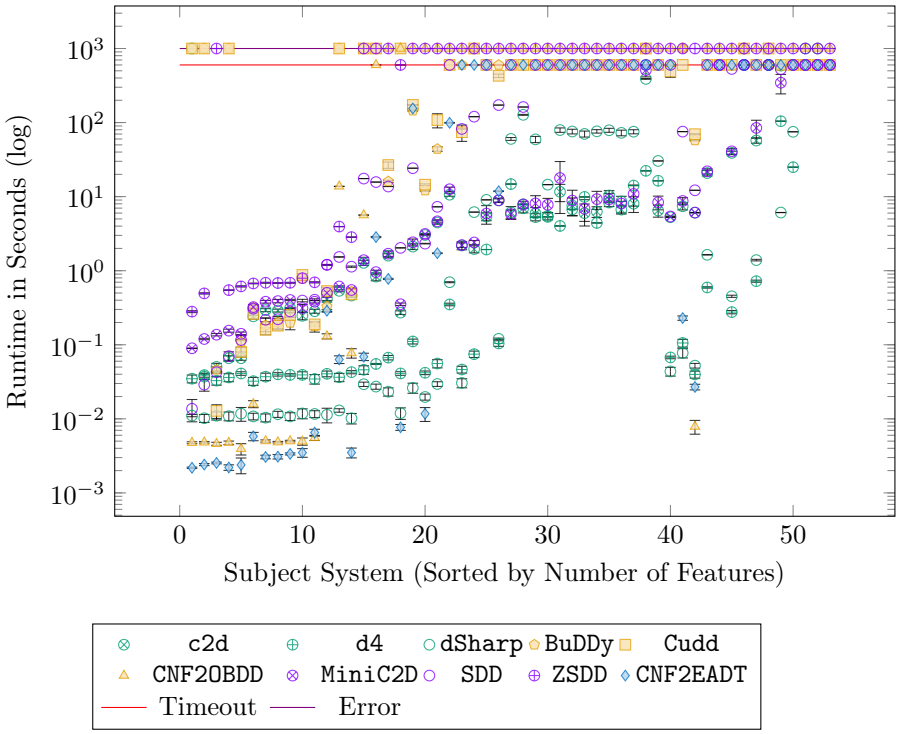

Fig. 11 Runtime Knowledge Compilers (Average & Standard Deviation)

- CPU: Intel(R) Xeon(R) CPU E5-260v3 with 2.40Ghz
- Overall RAM: 256 GB

During the measurements, no other major processes were performed on the machine. For each tool, we set the memory limit to 8 GB as we consider this a reasonable limit to expect for notebooks and PCs used in practice. Further, in preliminary experiments, we identified no substantial differences in measured runtimes when increasing the memory limit beyond 8 GB. For our empirical evaluation, we used a Python framework which internally uses the module `timeit` to measure runtimes.[15]

6.5 Results

Figure 11 shows the runtimes of all considered knowledge compilers on the 53 feature models. Each point on the x-axis corresponds to one feature model, which are sorted ascendingly by their number of features. The y-axis shows the runtime in seconds with a logarithmic scale. A mark on the red line indicates that the compiler hit the timeout for the respective feature model. The violet line indicates that a compiler terminated unexpectedly due to an error or hitting the memory limit.

[15] https://docs.python.org/3/library/timeit.html

Considering the different types of knowledge compilers, at least one of the d-DNNF compilers (`c2d`, `dSharp`, and `d4`) was able to compile 47 out of the 53 feature models. For the remaining six feature models, namely all (four) variants of *Linux*, *buildroot*, and *freetz*, all compilers for every target language hit the timeout or memory limit. At least one of the three SDD compilers, namely `MiniC2D`, `SDD`, and `ZSDD`, successfully compiled 46 feature models, failing to the same feature models as d-DNNFs and one additional feature model. The only EADT compiler `CNF2EADT` successfully compiled 25 feature models. The fastest BDD compiler `Cudd` was able to compile 18 feature models within the given timeout and memory limit.

Table 12 Overall Runtime Knowledge Compilers

Compiler	Successes (of 53)	Runtime (Sum over Successes)
`d4`	47	4.38 Min.
`dSharp`	47	21.8 Min.
`c2d`	44	14.3 Min.
`MiniC2D`	46	19.7 Min.
`SDD`	29	29.5 Min.
`ZSDD`	13	14.1 Min.
`CNF2OBDD`	16	19.7 Min.
`Cudd`	18	23.1 Min.
`BuDDy`	15	4.62 Min
`CNF2EADT`	25	4.54 Min.

Table 12 shows the number of successfully evaluated feature models and sum of runtimes for every evaluated knowledge compiler. The sum of runtimes corresponds to the runtimes over the feature models successfully evaluated by this compiler. The overall fastest d-DNNF compiler `d4` requires in sum 4.38 minutes for the 47 successfully evaluated feature models. For SDDs, the fastest compiler `MiniC2D` requires 19.7 minutes for the 46 feature models. `CNF2EADT` requires 4.54 minutes for the 25 feature models it successfully evaluated. `Cudd` successfully evaluated 16 of the feature models in overall 19.7 minutes.

6.6 Discussion

RQ1: How do knowledge compilers for different target languages perform on the task of compiling industrial feature models?
The majority of evaluated feature models can be compiled to at least one of the knowledge-compilation target languages. Still, 6 out of 53 feature models could not be compiled to any target language within the given timeout of ten minutes. Preliminary measurements outside of the experiment design indicated that even substantially increasing the timeout (e.g., to one hour) leads to very similar results.

Compiling to d-DNNF and SDD is applicable for many industrial feature models (88.7% and 86.8%, respectively). Compiling to BDD or EADT is only applicable (w.r.t. the runtime) for less complex feature models. The compilers for BDD or EADT only scaled to feature models for which compiling to d-DNNF requires less than a second.

The fastest compilers we identified for each target language are `d4` (d-DNNF), `MiniC2D` (SDD), `CNF2EADT` (EADT, only one), and `Cudd` (BDD). For each target language, neither compiler is faster than all other compilers of that target language on all considered feature models. Thus, even though the compilers listed above seem a promising choice for as compiler, it may still be beneficial to use different compilers depending on the input feature model.

Overall, our results indicate that knowledge compilation may be beneficial for feature-model analysis. The majority of feature models can be compiled within seconds to a target language that enables various tractable queries. To inspect the actual benefits, evaluating the performance of queries and comparing them with the current state of the art for feature-model analysis is essential as future work.

RQ2: Which knowledge-compilation target languages should be used for the different types of feature-model analyses?

Based solely on the compilation time, *d-DNNFs* are the most promising target language for SAT-, #SAT-, and AllSAT-based analyses. SDDs are competitive with d-DNNFs for many feature models and, thus, are also a considerable option. The benefits of target languages also depend on the runtime required for queries and, thus, longer compilation times (e.g., to SDD) could potentially be amortized if the queries are faster. However, compilers for the other considered target languages (i.e., BDD and EADT) fail to compute a target artifact for the majority of feature models within the given timeout.

Employing SDDs might also be valuable in the future as more queries are tractable for SDD than for d-DNNF. In particular, equivalence and sentential entailment can be checked in polynomial time with SDDs but not with d-DNNFs [89]. SDDs are also tractable for additional transformations compared to d-DNNFs, such as negation and bounded conjunction/disjunction [89]. While the same set of the feature-model analyses we collected are tractable for d-DNNF and SDD, future analyses or alternative ways to solve existing analyses may capitalize on the additional tractability of SDD.

Compiling to BDD may also be beneficial due to additional tractable queries. In particular, feature-model analyses dependent on algebraic operations, namely feature-model diffing and feature-model composition, are tractable for BDDs but not for EADT, d-DNNF, nor SDD.

It is important to note that due to availability of compilers, our empirical evaluation only covers four out of the ten identified target languages. While some target languages are not complete (i.e., cannot represent every propositional formula) and, thus, not necessarily suitable for feature-model

analysis, some of the six remaining target languages may be beneficial. Developing compilers for promising (w.r.t. tractable queries) target languages may be beneficial future work for feature-model analysis.

6.7 Threats to Validity

In this section, we discuss possible limitations to the results of our empirical evaluation and measures we have taken to reduce their impact.

Random Effects and Computational Bias

Single measurements for the same combination of feature model and compiler may vary due to random effects and computational bias. For instance, compilation with `c2d` partially uses randomness to create the initial vtree which the d-DNNF is built upon [99]. To reduce the impact of such effects, we performed ten repetitions for each measurement.

Compiler Parameters

Several considered compilers provide parameters (e.g., heuristics for selecting the branching variables) that potentially influence the compiler's runtime. Changing parameters may influence the performed measurements [108] and may even influence some conclusions. For such parameters, we always used the defaults as tweaking parameters would require further expertise. Hence, we expect default parameters to better reflect the usage of compilers for feature-model analysis in practice. Also, we consider parametrization of all compilers to be out of scope for this work. Still, we consider parameterizing the compilers as vital future work. A possible approach is to apply existing optimizers that more generically target parameter optimization for computationally complex problems [108, 109].

Formula Translations

The translation from feature model to formula is not necessarily unique; for example, there are alternative encodings for expressing alternatives [40]. Also, the selection of a CNF translation, such as Tseitin's transformation [110] or equivalence transformations, may influence the performance of considered tools [14]. Integrating more feature-model or CNF translations for every feature model would vastly increase the complexity of the empirical evaluation. Thus, we consider different translations as out of scope for this work but as relevant in the future.

Further, some CNF translations do not ensure equivalence of formulas but less restrictive properties, such as equisatisfiability. For such an equisatisfiable formula, other analyses such as #SAT may product incorrect results [14]. Such a transformation may also result in faulty knowledge compilation artifacts. However, for each feature model not already given as CNF, we used the FeatureIDE transformation, which ensures equivalence.

External Validity: Feature Models

Our empirical results cannot necessarily be transferred to other industrial feature models. However, we covered a wide range of feature models with regard to number of features (11–62,482), number of clauses (1–350,221), domains, and scalability of solvers and compilers (milliseconds vs. timeout for every compiler) [7].

External Validity: Compilers

It is possible that we missed some available knowledge compilers that would influence the results and implications of our empirical evaluation. In particular, we only found compilers for four out of the eleven considered knowledge-compilation target languages. To effectively find available compilers, we systematically considered a large variety of research on knowledge compilation. We surveyed 1,050 papers including the term *knowledge compilation* and extracted various target languages and compilers. Afterwards, we extended the list of compilers further by performing a GitHub search for each identified target language. A possible explanation for missing compilers targeting specific languages may be their tractability, completeness, and succinctness. We could not identify compiler for the four least tractable languages (considering feature-model analyses), namely Horn [80], EPCCL [81], DNF [82], and DNNF [84]. Horn is also incomplete (i.e., cannot represent all propositional formulas) which may be another reason for the missing compiler. We also did not find a compiler for MODS which is the least succinct target language.

7 Related Work

In this section, we discuss related work that (1) considers knowledge compilation in the feature-model domain and (2) surveys feature-model analyses. Further, we put our work into context of current research.

Knowledge Compilation on Feature Models

Binary decision diagrams have been considered for various feature-model analyses. Mendonca et al. [33] present several strategies for variable orderings based on the given feature model. Heradio et al. [32] use BDDs to compute core, dead, conditionally core, and conditionally dead features. In previous work [21], we analyzed the scalability of popular BDD libraries for compiling feature models. Our selection of BDD libraries in this work is based on the insights of the previous work [21]. Pohl et al. [34] compares the performance of several solving techniques, namely SAT solvers, BDDs, and CSP solvers for different feature-model analyses. Each of the listed publications considers only BDDs as target language and does not compare the scalability and capabilities (in terms of tractable queries) of BDDs to other target languages.

Sharma et al. [22] and Baital et al. [36] employ knowledge compilation to d-DNNF for uniform random sampling and t-wise sampling, respectively. However, their work is each limited to a single feature-model analysis and target

language while we consider a large variety of target languages and analyses. Bourhis et al. [111] perform several analyses on feature models using d-DNNFs. However, they do not consider any other target language. In previous publications, Kübler et al. [9] and we [7] used d-DNNF compilers as black-box #SAT solvers. In both publications, the knowledge-compilation artifact is not reused and, thus, the advantages of knowledge compilation are not used nor considered. Within another previous work [112], we employ d-DNNFs for model counting but consider no other analyses or target languages.

Voronov et al. [113] compare d-DNNFs (`c2d`) and BDDs (`BuDDy`) for enumerating valid configurations. Similar to our results, their empirical evaluation indicates that compiling to d-DNNF is substantially faster than compiling to BDD. In contrast to our work, the authors consider only one type of query (i.e., enumeration) and only two target languages. In previous work [114], we evaluate several model counters (or #SAT solvers) including some knowledge compilers capable of counting. Thereby, we exclude target languages intractable of counting and measure the runtimes required for counting instead of only compilation.

Krieter et al. [51] employ modal implication graphs to accelerate decision propagation (cf. Section 4.1). A modal implication graph encodes an underlying CNF as a directed graph indicating dependencies between features. Hereby, a binary clause $A \vee B$ is represented by two strong edges (i.e., $\neg A \Rightarrow B$ and $\neg B \Rightarrow A$) between the corresponding literals indicating the implications. Clauses with more than two literals (e.g. $A \vee B \vee C$) are represented by weak edges that indicate a possible implication (e.g., $\neg A \Rightarrow B \vee C$). While modal implications graphs reduce the number of required SAT calls and, thus, accelerate decision propagation in practice, none of the considered feature-model analysis is tractable for them. Krieter et al. [51] also consider an extension of modal implications graphs indicating all core and dead features. With this extension, feature model consistency and feature consistency are tractable for modal implication graphs. However, the set of core and dead features alone makes both analyses tractable and, thus, we do not consider pure modal implications graphs as a knowledge-compilation target language.

Surveys on Feature-Model Analyses

Benavides et al. [2] gathered various feature-model analyses depending on different computational problems (e.g., SAT, #SAT, or AllSAT). While giving general ideas on how to compute results for some specified analyses, the authors do not provide an assessment on the computational complexity regarding the number of queries. Further, we provide 25 additional analyses.

Galindo et al. [115] conduct a systematic literature survey focusing on publications dealing with automated analysis of feature models. However, they do not focus on concrete analyses but rather on publication metrics, such as popularity of venues and background of authors.

In previous work [26], we presented several analyses dependent on feature-model counting, fully subsuming the list of analyses of previous surveys on

feature-model counting [9, 50]. There, we categorized each analysis by the type of computation, namely cardinality of feature model, features, and partial configurations. In contrast to this previous work, we now analyze the number of required queries and extend the list of analyses with analyses which depend on other computational problems.

In other previous work [1], we classified analysis strategies for product lines. This classification targets the application domain (e.g., product-based vs. family-based analysis) in contrast to showcasing underlying computational problems. Further, with the previous survey [1] we did not elaborate on concrete analyses but rather on more abstract analysis classes.

Various other surveys on product lines consider feature-model analyses to some degree [116–120]. While some publications provide rough ideas on the complexity of respective analyses, neither work systematically analyzes the complexities. In contrast, we provide a worst-case complexity for every collected feature-model analysis.

8 Conclusion

In our survey, we show that many feature-model analyses depend on numerous queries on the exact same feature model. Further, many analyses can be applied repeatedly for the same feature model, for instance when interactively deriving configurations. The potentially high number of required queries motivates the use of knowledge compilation. With knowledge compilation, the original feature model is translated to a target language that enables faster querying, potentially amortizing the initial cost of compilation.

Available target languages allow fast queries for different sets of analyses. Finding a sweet spot between tractable queries and compilation is essential to get full benefits from employing knowledge compilation. d-DNNFs and SDDs enable polytime queries for a variety of feature-model analyses and scale to the majority of considered industrial feature models. Hence, both target languages seem promising to apply. Other formats, such as BDDs and EADT, require substantially more time to compile, hitting the timeout for the majority of feature models. Nevertheless, if a feature-model analysis of interest requires complex queries such as algebraic computations, BDD and EADT still yield benefits over d-DNNF and SDD.

9 Future Work

Our results on the applicability of knowledge-compilation target languages and the scalability of available compilers indicate the potential of knowledge compilation. However, the benefits in practice also depend on the runtimes of querying the knowledge-compilation artifact. If the queries are too slow, the effort to compile the original formula might not amortize. To quantify the actual benefits, the efficiency of queries should be examined and compared with state-of-the-art solutions.

For many knowledge-compilation target languages we collected, we were not able to find any publicly available compilers. As each of the considered target languages enables some queries relevant for feature-model analysis, developing compilers may be beneficial.

Feature-model specific optimizations may improve the runtimes of knowledge compilers. In our work, we used the compilers as black boxes without any parameterization. Identifying and eventually automatically selecting parameters may increase the performance and, thus, could be valuable for future work.

Declarations

Conflict of Interests

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Data Availability

The datasets generated during and/or analyzed during the current study are available in the replication repository, https://github.com/SoftVarE-Group/kc-for-fmanalysis-evaluation/.

References

[1] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Computing Surveys (CSUR) **47**(1), 6–1645 (2014)

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

[3] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, (2005)

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

[5] McGregor, J.: Testing a Software Product Line. In: Testing Techniques in Software Engineering, pp. 104–140. Springer, (2010)

[6] Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, (2013)

[7] Sundermann, C., Thüm, T., Schaefer, I.: Evaluating #SAT Solvers on Industrial Feature Models. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, (2020)

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

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

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

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

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

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

[14] Kuiter, E., Krieter, S., Sundermann, C., Thüm, T., Saake, G.: Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), pp. 110–111013. ACM, (2022)

[15] Oh, J., Gazzillo, P., Batory, D.: t-wise Coverage by Uniform Sampling. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 84–87. ACM, (2019)

[16] Oh, J., Gazzillo, P., Batory, D., Heule, M., Myers, M.: Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The University of Texas at Austin, Department of Computer Science (2019)

[17] Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: Is There a Mismatch between Real-World Feature Models and Product-Line Research? In: Tichy, M., Bodden, E., Kuhrmann, M., Wagner, S., Steghöfer, J. (eds.) Proc. Software Engineering (SE), pp. 53–54. Gesellschaft für Informatik, (2018)

[18] Valiant, L.G.: The Complexity of Enumeration and Reliability Problems. SIAM J. on Computing **8**(3), 410–421 (1979)

[19] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Mastering Software Variability with FeatureIDE. Springer, (2017)

[20] Hentze, M., Pett, T., Thüm, T., Schaefer, I.: Hyper Explanations for Feature-Model Defect Analysis. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, (2021)

[21] Heß, T., Sundermann, C., Thüm, T.: On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 131–135. ACM, (2021)

[22] Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge Compilation Meets Uniform Sampling. In: Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 620–636. EasyChair, (2018)

[23] Drechsler, R., Becker, B.: Binary Decision Diagrams: Theory and Implementation. Springer, (1998)

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

[25] Darwiche, A., Marquis, P.: A Knowledge Compilation Map. J. Artificial Intelligence Research (JAIR) **17**(1), 229–264 (2002)

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

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

[28] Fernández-Amorós, D., Heradio, R., Cerrada, J.A., Cerrada, C.: A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. IEEE Trans. on Software Engineering (TSE) **40**(9), 895–910 (2014)

[29] Krieter, S., Schröter, R., Thüm, T., Saake, G.: An Efficient Algorithm for Feature-Model Slicing. Technical Report FIN-001-2016, University of Magdeburg (2016)

[30] Segura, S.: Automated Analysis of Feature Models Using Atomic Sets. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), vol. 2, pp. 201–207. IEEE, (2008)

[31] El-Sharkawy, S., Krafczyk, A., Schmid, K.: An Empirical Study of Configuration Mismatches in Linux. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 19–28. ACM, (2017)

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

[33] Mendonça, M., Wąsowski, A., Czarnecki, K., Cowan, D.: Efficient Compilation Techniques for Large Scale Feature Models. In: Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE), pp. 13–22. ACM, (2008)

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

[35] Darwiche, A.: SDD: A New Canonical Representation of Propositional Knowledge Bases, pp. 819–826. AAAI Press, (2011)

[36] Baranov, E., Legay, A., Meel, K.S.: Baital: An Adaptive Weighted Sampling Approach for Improved t-Wise Coverage. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 1114–1126. ACM, (2020)

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

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

[39] Heradio, R., Fernández-Amorós, D., Mayr-Dorn, C., Egyed, A.: Supporting the Statistical Analysis of Variability Models. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 843–853. IEEE, (2019)

[40] Karpiński, M., Piotrów, M.: Encoding Cardinality Constraints Using Multiway Merge Selection Networks. Constraints **24**, 234–251 (2019)

[41] Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: 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), pp. 291–302. ACM, (2017)

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

[43] Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 502–518. Springer, (2004)

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

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

[46] Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: Fast d-dnnf compilation with sharpsat. In: Kosseim, L., Inkpen, D. (eds.) Advances in Artificial Intelligence, pp. 356–361. Springer, (2012)

[47] Büning, H.K., Lettmann, T.: Propositional Logic: Deduction and Algorithms vol. 48. Cambridge University Press, (1999)

[48] Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. Technical report, Cincinnati University (1996)

[49] Valiant, L.G.: The Complexity of Computing the Permanent. Theoretical Computer Science **8**(2), 189–201 (1979)

[50] Heradio, R., Fernández-Amorós, D., Cerrada, J.A., Abad, I.: 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), 1177–1204 (2013)

[51] Krieter, S., Thüm, T., Schulze, S., Schröter, R., Saake, G.: Propagating Configuration Decisions with Modal Implication Graphs. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 898–909. ACM, (2018)

[52] Oh, J., Gazzillo, P., Batory, D., Heule, M., Myers, M.: Scalable Uniform Sampling for Real-World Software Product Lines. Technical Report

TR-20-01, The University of Texas at Austin, Department of Computer Science (2020)

[53] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. Science of Computer Programming (SCP) **79**(0), 70–85 (2014)

[54] Zhang, W., Zhao, H., Mei, H.: A Propositional Logic-Based Method for Verification of Feature Models. In: Proc. Int'l Conf. on Formal Engineering Methods (ICFEM), pp. 115–130. Springer, (2004)

[55] Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. Empirical Software Engineering (EMSE) **24**(2), 674–717 (2019)

[56] Thüm, T., Batory, D., Kästner, C.: Reasoning About Edits to Feature Models. In: Proc. Int'l Conf. on Software Engineering (ICSE), pp. 254–264. IEEE, (2009)

[57] Janota, M., Lynce, I., Marques-Silva, J.: Algorithms for Computing Backbones of Propositional Formulae. Ai Communications **28**(2), 161–177 (2015)

[58] Krieter, S., Thüm, T., Schulze, S., Schröter, R., Saake, G.: Propagating Configuration Decisions with Modal Implication Graphs. In: Proc. Software Engineering (SE), pp. 77–78. Gesellschaft für Informatik, (2019)

[59] Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), pp. 144–155. ACM, (2016)

[60] Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In: Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 16–1166. ACM, (2013)

[61] Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 638–652. Springer, (2011)

[62] Johansen, M.F., Haugen, Ø., Fleurey, F.: An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In: Proc. Int'l

Systems and Software Product Line Conf. (SPLC), pp. 46–55. ACM, (2012)

[63] Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In: Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 269–284. Springer, (2012)

[64] Kowal, M., Schulze, S., Schaefer, I.: Towards Efficient SPL Testing by Variant Reduction. In: Proc. Int'l Workshop on Variability and Composition (VariComp), pp. 1–6. ACM, (2013)

[65] Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 196–210. Springer, (2010)

[66] Acher, M., Heymans, P., Collet, P., Quinton, C., Lahire, P., Merle, P.: Feature Model Differences. In: Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE), pp. 629–645. Springer, (2012)

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

[68] Bagheri, E., Noia, T.D., Gasevic, D., Ragone, A.: Formalizing Interactive Staged Feature Model Configuration. J. Software: Evolution and Process **24**(4), 375–400 (2012)

[69] Clements, P.C., McGregor, J.D., Cohen, S.G.: The Structured Intuitive Model for Product Line Economics (SIMPLE). Technical report, Carnegie-Mellon University (2005)

[70] Cohen, S.: Predicting When Product Line Investment Pays. Technical report, Carnegie-Mellon University (2003)

[71] Chen, S., Erwig, M.: Optimizing the Product Derivation Process. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 35–44. IEEE, (2011)

[72] Mazo, R., Dumitrescu, C., Salinesi, C., Diaz, D.: Recommendation Heuristics for Improving Product Line Configuration Processes. In: Recommendation Systems in Software Engineering, pp. 511–537. Springer, (2014)

[73] Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding Near-Optimal Configurations in Product Lines by Random Sampling. In: Proc. Int'l Symposium on Foundations of Software Engineering (FSE), pp. 61–71

(2017)

[74] Aziz, R.A., Chu, G., Muise, C., Stuckey, P.: #∃ SAT: Projected Model Counting. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 121–137. Springer, (2015)

[75] Acher, M., Collet, P., Lahire, P., France, R.B.: Slicing Feature Models. In: Proc. Int'l Conf. on Automated Software Engineering (ASE), pp. 424–427. IEEE, (2011)

[76] Krieter, S., Schröter, R., Thüm, T., Fenske, W., Saake, G.: Comparing Algorithms for Efficient Feature-Model Slicing. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 60–64. ACM, (2016)

[77] Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting Treewidth for Projected Model Counting and Its Limits. In: Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT), pp. 165–184. Springer, (2018)

[78] Acher, M., Collet, P., Lahire, P., France, R.B.: Comparing Approaches to Implement Feature Model Composition. In: Proc. Europ. Conf. on Modelling Foundations and Applications (ECMFA), pp. 3–19. Springer, (2010)

[79] Rice, M., Kulhari, S.: A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction. Technical report, University of California, Riverside (2008)

[80] Dowling, W.F., Gallier, J.H.: Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. J. Logic Programming **1**(3), 267–284 (1984)

[81] Wang, J., Gu, W., Yin, M., Wang, D.: MCN and MO: Two Heuristic Strategies in Knowledge Compilation Using Extension Rule. In: International Conference on Signal Processing Systems (ICSPS), pp. 389–393 (2009)

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

[83] Pipatsrisawat, K., Darwiche, A.: New Compilation Languages Based on Structured Decomposability. In: Proc. Conf. on Artificial Intelligence (AAAI), vol. 8, pp. 517–522. AAAI Press, (2008)

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

[85] Darwiche, A.: New Advances in Compiling CNF to Decomposable Negation Normal Form. In: Proc. Europ. Conf. on Artificial Intelligence, pp. 318–322. IOS Press, (2004)

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

[87] Muise, C., McIlraith, S., Beck, J.C., Hsu, E.: Fast d-DNNF Compilation with sharpSAT. In: Proc. Conf. on Artificial Intelligence (AAAI). AAAI Press, (2010)

[88] Koriche, F., Lagniez, J.-M., Marquis, P., Thomas, S.: Knowledge Compilation for Model Counting: Affine Decision Trees. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI). AAAI Press, (2013)

[89] Van den Broeck, G., Darwiche, A.: On the Role of Canonicity in Knowledge Compilation. Proc. Conf. on Artificial Intelligence (AAAI) **29**(1) (2015)

[90] Nishino, M., Yasuda, N., Minato, S.-i., Nagata, M.: Zero-Suppressed Sentential Decision Diagrams. In: Proc. Conf. on Artificial Intelligence (AAAI), pp. 1058–1066. AAAI Press, (2016)

[91] Oztok, U., Darwiche, A.: An Exhaustive DPLL Algorithm for Model Counting **62**(1), 1–32 (2018)

[92] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Computers **C-35**(8), 677–691 (1986)

[93] Jha, A., Suciu, D.: Knowledge compilation meets database theory: Compiling queries to decision diagrams. In: Proceedings of the 14th International Conference on Database Theory (ICDT), pp. 162–173. ACM, (2011)

[94] Minato, S.-i.: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. Anual Conf. on Design Automation (DAC), pp. 272–277. ACM, (1993)

[95] Razgon, I.: On Oblivious Branching Programs With Bounded Repetition That Cannot Efficiently Compute Cnfs of Bounded Treewidth. Theory of Computing Systems **61**(3), 755–776 (2017)

[96] Vinkhuijzen, L., Laarman, A.: The Power of Disjoint Support Decompositions in Decision Diagrams. In: NASA Formal Methods Symposium (NFM), pp. 790–799. Springer, (2022)

[97] van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: Combining Reduction Rules from Different Decision Diagram Types. In: Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD), pp. 108–115 (2017). IEEE

[98] Bryant, R.E.: Binary Decision Diagrams. In: Handbook of Model Checking, pp. 191–217. Springer, (2018)

[99] Oztok, U., Darwiche, A.: On Compiling CNF into Decision-DNNF. In: Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP), pp. 42–57. Springer, (2014)

[100] Pett, T., Krieter, S., Thüm, T., Lochau, M., Schaefer, I.: AutoSMP: An Evaluation Platform for Sampling Algorithms. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 41–44. ACM, (2021)

[101] Nieke, M., Mauro, J., Seidl, C., Thüm, T., Yu, I.C., Franzke, F.: Anomaly Analyses for Feature-Model Evolution. In: Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE), pp. 188–201. ACM, (2018)

[102] Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product Sampling for Product Lines: The Scalability Challenge. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC), pp. 78–83. ACM, (2019)

[103] Oztok, U., Darwiche, A.: A Top-Down Compiler for Sentential Decision Diagrams. In: Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI), pp. 3141–3148. AAAI Press, (2015)

[104] Somenzi, F.: Efficient Manipulation of Decision Diagrams. Int'l J. Software Tools for Technology Transfer (STTT) **3**(2), 171–181 (2001)

[105] Janssen, G.: A Consumer Report on BDD Packages. In: Proc. Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 217–222. IEEE, (2003)

[106] Toda, T., Soh, T.: Implementing Efficient All Solutions SAT Solvers. ACM J. of Experimental Algorithmics (JEA) **21**(1), 1–12111244 (2016)

[107] Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I.: Stability of Product-Line Sampling in Continuous Integration. In: Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS). ACM, (2021)

[108] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based

optimization for general algorithm configuration. In: Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5, pp. 507–523 (2011). Springer

[109] Coy, S.P., Golden, B.L., Runger, G.C., Wasil, E.A.: Using Experimental Design to Find Effective Parameter Settings for Heuristics. Journal of Heuristics **7**, 77–97 (2001)

[110] Tseytin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer, (1983)

[111] Bourhis, P., Duchien, L., Dusart, J., Lonca, E., Marquis, P., Quinton, C.: Reasoning on Feature Models: Compilation-Based vs. Direct Approaches. Technical Report arXiv:2302.06867, Cornell University Library (2023)

[112] Sundermann, C., Raab, H., Heß, T., Thüm, T., Schaefer, I.: Exploiting d-DNNFs for Repetitive Counting Queries on Feature Models. Technical Report arXiv:2303.12383, Cornell University Library (2023)

[113] Voronov, A., Åkesson, K., Ekstedt, F.: Enumeration of Valid Partial Configurations. In: Proc. Configuration Workshop (ConfWS), vol. 755, pp. 25–31. ceur-ws.org, (2011)

[114] Sundermann, C., Heß, T., Nieke, M., Bittner, P.M., Young, J.M., Thüm, T., Schaefer, I.: Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. Empirical Software Engineering (EMSE) **28** (2023)

[115] Galindo, J.A., Benavides, D., Trinidad, P., Gutiérrez-Fernández, A.-M., Ruiz-Cortés, A.: Automated Analysis of Feature Models: Quo Vadis? Computing **101**(5), 387–433 (2019)

[116] Janota, M., Kiniry, J., Botterweck, G.: Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick (2008)

[117] Lutz, R.: Survey of Product-Line Verification and Validation Techniques. Technical Report 2014/41221, NASA, Jet Propulsion Laboratory (2007)

[118] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature Diagrams: A Survey and a Formal Semantics. In: Proc. Int'l Conf. on Requirements Engineering (RE), pp. 136–145. IEEE, (2006)

[119] Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A Classification of Product Sampling for Software Product Lines. In: Proc. Int'l Systems and Software Product Line Conf. (SPLC),

pp. 1–13. ACM, (2018)

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