Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses

Elias Kuiter
Otto-von-Guericke-University
Magdeburg, Germany
kuiter@ovgu.de

Sebastian Krieter
University of Ulm
Ulm, Germany
sebastian.krieter@uni-ulm.de

Chico Sundermann
University of Ulm
Ulm, Germany
chico.sundermann@uni-ulm.de

Thomas Thüm
University of Ulm
Ulm, Germany
thomas.thuem@uni-ulm.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

ABSTRACT

Feature modeling is widely used to systematically model features of variant-rich software systems and their dependencies. By translating feature models into propositional formulas and analyzing them with solvers, a wide range of automated analyses across all phases of the software development process become possible. Most solvers only accept formulas in conjunctive normal form (CNF), so an additional transformation of feature models is often necessary. However, it is unclear whether this transformation has a noticeable impact on analyses. In this paper, we compare three transformations (i.e., distributive, Tseitin, and Plaisted-Greenbaum) for bringing feature-model formulas into CNF. We analyze which transformation can be used to correctly perform feature-model analyses and evaluate three CNF transformation tools (i.e., FeatureIDE, KConfigReader, and Z3) on a corpus of 22 real-world feature models. Our empirical evaluation illustrates that some CNF transformations do not scale to complex feature models or even lead to wrong results for model-counting analyses. Further, the choice of the CNF transformation can substantially influence the performance of subsequent analyses.

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; • **Theory of computation** → **Automated reasoning**; • **Computing methodologies** → *Representation of Boolean functions*; • **Hardware** → *Theorem proving and SAT solving*.

KEYWORDS

Feature Modeling, Automated Reasoning, Conjunctive Normal Form

ACM Reference Format:
Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556938

1 INTRODUCTION

Many software systems in today's industry can be diversely configured to serve specific customer needs [11, 48, 115]. For example, the Linux kernel has more than 15,000 configuration options or *features* [41] and an unknown astronomical number of valid configurations [89, 110]. For such variant-rich software systems or *software product lines* [4, 95], it is necessary to model features and their dependencies systematically [9]. *Feature models* [8, 28, 54] are widely used for this task [12, 27], as they facilitate communication between stakeholders and enable automated analysis of the configuration space [2, 9, 10, 79, 96, 124].

Satisfiability (SAT) solvers [34, 53, 80] search for satisfying assignments of propositional formulas and are routinely used for the automated analysis of feature models [9, 28, 71, 79]. As feature models play a role in all phases of the software development process, SAT solving enables a wide range of automated analyses in the context of interactive configuration [50, 64], anomaly detection [9, 86, 103] and explanation [36, 60], evolution [61, 116], modularization [62, 101], testing [57, 90], static code analysis [18, 72], type checking [5, 56, 58], model checking [6, 97] and formal verification [118]. Similarly, *model-counting (#SAT) solvers* [38, 82, 105, 119] count satisfying assignments of propositional formulas and also empower numerous product-line analyses [109], including feature prioritization [109], detecting errors [67, 109], various economical estimations [19, 26, 37], and uniform random sampling [83, 89].

To analyze feature models using solvers, they must be translated into propositional formulas and typically even into *conjunctive normal form (CNF)*. For feature models with Boolean (i.e., a feature is either selected or deselected) or numerical features, translations into propositional formulas are well-known [4, 8, 83, 100]. However, the industry-standard format [22, 47, 49, 51, 53, 98] for exchanging and analyzing propositional formulas with solvers, *DIMACS* [33], can only represent formulas in CNF. Thus, a subsequent transformation into CNF is commonly applied, which is not obvious for feature models with complex feature dependencies [59]. In many papers on feature-model and variability analysis, this step, although necessary, is not mentioned [35, 62, 68, 69, 103, 108] or discussed only superficially [8, 9, 16, 37, 45, 64, 71, 96, 116], for example by only referring to Tseitin [120] without specifying more details on the transformation. In addition, many tools for automated feature-model extraction (e.g., LVAT [106], UNDERTAKER [112], and KCONFIGREADER [35, 55]) or analysis (e.g., FEATUREIDE [78, 117] and KERNELHAVEN [65]) implement a CNF transformation, for which we were unable to find any documentation (KCLAUSE [88] being the only exception). Consequently, the details of the chosen CNF transformation are lost in several benchmarks for feature-model analysis [13, 59]. However, in industry collaborations, we repeatedly observed that using different CNF transformations may affect the efficiency and results of analyses based on SAT and #SAT solvers, which indicates that the selection of a CNF transformation is relevant for practitioners. In addition, ignoring this issue may pose a potential threat to validity for research evaluations [107].

In this paper, we describe, compare, and evaluate several state-of-the-art techniques for transforming feature-model formulas into CNF. We aim to determine whether the CNF transformation's impact on SAT- and #SAT-based feature-model analyses is noticeable; that is, whether it may fail to produce a result in reasonable time, significantly affect the runtime of analyses, or even cause incorrect analysis results. Thus, we are the first to systematically investigate whether the chosen CNF transformation influences the work of practitioners or threatens the validity of research evaluations. We make the following contributions in this paper:

- We propose a taxonomy of five properties that CNF transformations may fulfill, and characterize its connection to selected feature-model analyses.
- We describe three common CNF transformation algorithms and classify their theoretical capabilities using our taxonomy of transformation properties.
- We evaluate and compare the efficiency and correctness of three CNF transformation tools commonly used for feature-model analyses on a corpus of 22 real-world feature models.

In summary, we aim to raise the awareness of researchers and practitioners regarding the impact that choosing a CNF transformation can have on feature-model and product-line analyses.

2 BACKGROUND

In the following, we describe our notion of propositional formulas, how to represent feature models as formulas, and how to analyze feature-model formulas with solvers.

2.1 Propositional Logic

We briefly recapitulate the basics of propositional (i.e., Boolean) logic, which we use to represent feature models. A propositional formula ϕ is a *Boolean variable* x, a *negation* $(\neg\phi)$ of a formula ϕ or a connection of two formulas ϕ and ψ by one of the connectors *and* $(\phi \wedge \psi)$, *or* $(\phi \vee \psi)$, *implies* $(\phi \rightarrow \psi)$, or *biimplies* $(\phi \leftrightarrow \psi)$. A variable x and its negation $\neg x$ are also referred to as *literals*. Implications $\phi \rightarrow \psi$ and biimplications $\phi \leftrightarrow \psi$ can be equivalently expressed as $\neg\phi \vee \psi$ and $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$, respectively [23], which is why we focus on expressions using only the *and* and *or* connectors. We refer to the set of all possible formulas as Φ. A propositional formula $\phi \in \Phi$ is in *conjunctive normal form* (CNF) [23], also known as *clause form* [20, 49, 87, 93], if it is a conjunction of *clauses*, which are disjunctions of literals (i.e., ϕ is of the form $(x_{1,1} \vee \ldots \vee x_{1,m_1}) \wedge \ldots \wedge (x_{n,1} \vee \ldots \vee x_{n,m_n})$, where all $x_{i,j}$ are literals). Conjunctions and disjunctions can have an arbitrary number of arguments ≥ 1, so $x_1 \vee x_2$ (one conjunct) and $x_1 \wedge x_2$ (two conjuncts with one disjunct each) are in CNF, too.

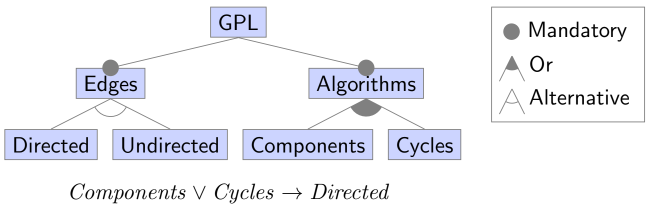

Figure 1: Example feature diagram for a graph product line.

We evaluate the truth value (i.e., $\top$ for *true* or $\bot$ for *false*) of a formula by assigning a truth value to each of its variables. For brevity, we do not consider a formula and its variables independently (e.g., as a tuple (ϕ, Var_ϕ)), but assume that the set $Var(\phi)$ contains all Boolean variables assignable in ϕ. An assignment for a formula ϕ then consists of a subset of variables $A \subseteq Var(\phi)$, each of which is implicitly set to $\top$. All other variables (i.e., those in $Var(\phi) \setminus A$) are implicitly set to $\bot$. Given a formula ϕ and a corresponding assignment A, we can evaluate $\phi(A)$ using the usual rules of Boolean algebra [23]. We denote the set of all satisfying assignments for a formula ϕ as $[\![\phi]\!] = \{A \mid A \subseteq Var(\phi), \phi(A) = \top\}$.

Finding and counting satisfying assignments are fundamental problems in propositional logic, which are known to be NP- and #P-complete, respectively. SAT and #SAT solvers are highly-optimized tools that automate the solving of these problems [38, 53]. A SAT solver [34, 80] is a program that determines whether a given formula ϕ is satisfiable (i.e., $[\![\phi]\!] \neq \emptyset$), and it usually returns some satisfying assignment from $[\![\phi]\!]$ (e.g., using DPLL [30, 31]). A #SAT solver [82, 105, 119] determines the actual *number* of satisfying assignments (i.e., the model count $|[\![\phi]\!]|$) by building an intermediate representation (e.g., a d-DNNF [82]). In the last decades, many problems from different domains (e.g., feature modeling) have been reduced to finding or counting satisfying assignments for propositional formulas [9, 115].

2.2 Feature Modeling

Feature modeling is a core activity for the development and management of software product lines and other variant-rich software [4, 27, 95]. A feature model defines the features of a product line as well as their interdependencies. Each feature describes a unique characteristic of the product line, which may or may not occur in a specific product [4]. To configure a product line, a user selects a subset of features from the feature model (i.e., a configuration), which can then be used to derive a particular product.

For automated analyses, we can represent the dependencies of a feature model as a propositional formula ϕ over the set of features F (i.e., $F = Var(\phi)$). A configuration is then represented by an assignment $C \subseteq F$ for the feature-model formula ϕ. If an assignment C satisfies the feature dependencies given by ϕ (i.e., $C \in [\![\phi]\!]$), we call the corresponding configuration *valid*, which means that it can be used to derive a product. That is, $[\![\phi]\!]$ represents the product line's problem space (i.e., set of valid configurations).

Besides propositional formulas, there are other representations of (Boolean) feature models that can all be translated into the same logical representation [4, 8]. One popular representation is the feature diagram [54], which arranges the features of a product line in a hierarchical tree structure that implicitly defines constraints between features, such as *optional/mandatory*, *or*, and *alternative* relationships. In Figure 1, we show an example feature model for the graph product line [74], represented as a feature diagram. In this product line, we have seven features to implement different kinds of edges in a graph library (e.g., directed and undirected edges) as well as different algorithms on such graphs (e.g., calculating strongly connected components or cycles). The tree notation already imposes some constraints on the problem space. For example, *Edges* and *Algorithms* are mandatory features which cannot be deselected. In addition, the features below *Edges* are alternative (i.e., exactly one must be selected) and those below *Algorithms* are in an or-relationship (i.e., at least one must be selected). With an additional cross-tree constraint below the feature tree, we ensure that if the user chooses any algorithm, directed edges are used.

Any feature diagram can be translated into a propositional formula [8]. For instance, the diagram in Figure 1 corresponds to the following propositional formula, which is not in CNF yet due to the additional cross-tree constraint:

$$\begin{aligned}\phi_{GPL} = {} & GPL \land Edges \land Algorithms \land (Directed \lor Undirected)\\ & \land (\neg Directed \lor \neg Undirected) \land (Components \lor Cycles)\\ & \land (\neg(Components \lor Cycles) \lor Directed)\end{aligned}$$

There exist three satisfying assignments for this formula:

$$\begin{aligned}[\![\phi_{GPL}]\!] = \{ & \{GPL, Edges, Alg., Dir., Components\},\\ & \{GPL, Edges, Alg., Dir., Cycles\},\\ & \{GPL, Edges, Alg., Dir., Components, Cycles\}\}\end{aligned}$$

Together, these comprise the problem space of this product line.

2.3 Feature-Model Analyses

Feature models can be analyzed to infer knowledge about implicit feature dependencies and to find anomalies in its described problem space [9, 28, 71, 79]. Almost all tool support for planning, configuring, developing, and testing product lines requires information from these automated analyses [5, 6, 9, 18, 36, 50, 56–58, 60–62, 64, 72, 86, 90, 97, 101, 103, 116, 118]. In the following, we describe several well-known basic feature-model analyses that are based on using a SAT or #SAT solver [4, 9, 109]. For the sake of brevity, we do not consider more complex analyses, because these analyses use solvers in a similar fashion [9, 109].

2.3.1 Void Feature Models. A feature model is *void* if it has no valid configurations because of some contradictory constraints [9]. This situation is undesirable, as users will never be able to derive a product. For instance, the feature model in Figure 1 is *not void* as it has three valid configurations. Whether a feature model is void can be determined by analyzing whether it has valid configurations. Given a feature-model formula ϕ, we define the void analysis as:

$$void(\phi) \text{ if and only if } [\![\phi]\!] = \varnothing$$

If ϕ is in CNF, we can implement this analysis by querying a SAT solver for ϕ. If ϕ is unsatisfiable, the feature model is void.

2.3.2 Dead and Core Features. A *dead feature* is not contained in any valid configuration of a given feature model [9]. In contrast, a *core feature* is contained in all valid configurations [9]. Typically, it is desirable to know which features are dead and core, for example to identify features that are temporarily disabled [44], or to identify common parts of all variants in a product line. For example, in Figure 1, the features *GPL*, *Edges*, *Algorithms*, and *Directed* are core. Feature *Undirected* is dead, because it is alternative to feature *Directed*, which itself is required by all algorithms. For a feature-model formula ϕ and a feature f, we define:

$$\begin{aligned}dead(\phi, f) &\text{ if and only if } [\![\phi \land f]\!] = \varnothing\\ core(\phi, f) &\text{ if and only if } [\![\phi \land \neg f]\!] = \varnothing\end{aligned}$$

These analyses test whether the selection (i.e., $\phi \land f$) or deselection (i.e., $\phi \land \neg f$) of f leads to a contradiction, which means that f cannot be selected or deselected, respectively. If ϕ is in CNF, we can use a SAT solver to implement these analyses by testing whether the formulas $\phi \land f$ and $\phi \land \neg f$ have satisfying assignments.

2.3.3 Feature-Model and Feature Cardinalities. Cardinality analyses count the number of valid configurations for a feature model (i.e., *feature-model cardinality*) or a specific feature (i.e., *feature cardinality*) [109]. For instance, the feature model in Figure 1 has a feature-model cardinality of 3, because it has three valid configurations. The feature cardinality of the feature *Components* is 2, because it occurs in two valid configurations. For a feature-model formula ϕ and feature f, we define feature-model cardinality $\#fm(\phi)$ and feature cardinality $\#f(\phi, f)$:

$$\begin{aligned}\#fm(\phi) &= |[\![\phi]\!]|\\ \#f(\phi, f) &= |[\![\phi \land f]\!]|\end{aligned}$$

Such cardinality analyses have several applications: For instance, we can determine the variability factor of a feature model, which measures how restrictive the model is by considering the ratio $\#fm(\phi)/2^{|Var(\phi)|}$. In our example in Figure 1, the variability factor is $3/128 \approx 2.34\%$, so only a small fraction of configurations is indeed valid. Another application is determining the commonality of a feature, which measures the relevance of a feature by considering the ratio $\#f(\phi,f)/\#fm(\phi)$. For example, the feature *Components* in Figure 1 has commonality $2/3$, while *Directed* has commonality 1.

While, in theory, these cardinality analyses can be implemented as repeated SAT calls excluding previously found configurations, a #SAT solver is substantially more efficient for this task [24]. For calculating the feature-model cardinality of a feature-model formula ϕ that is in CNF, we can query a #SAT solver for ϕ. For calculating the feature cardinality for a feature f, we instead query for $\phi \land f$.

3 CNF TRANSFORMATION PROPERTIES

To compute feature-model analyses using solvers, the analyzed feature-model formula must first be transformed into conjunctive normal form (CNF) [49, 53, 98]. As feature models can contain arbitrarily complex cross-tree constraints, a direct CNF encoding is usually not feasible [59], so transformation algorithms are needed. In addition, to ensure correct analysis results, it is important that the transformed formula preserves certain properties of the input formula [88]. Thus, we distinguish classes of CNF transformations by presenting a formal taxonomy of five properties for CNF transformations (cf. Section 3.1). Then, we show how these properties can serve as a tool to analyze the correctness of feature-model analyses in the presence of a CNF transformation (cf. Section 3.2).

3.1 Taxonomy of Transformation Properties

To describe our taxonomy of transformation properties, we formally define CNF transformations [43, 98], which are also known as CNF conversions [25, 49], encodings [52, 98], or translations [20, 93].

Definition 3.1. A *CNF transformation* is a function $\theta\colon \Phi \rightarrow \Phi$ that maps one formula to another such that $\theta(\phi)$ is in CNF (i.e., a conjunction of disjunctions of literals) and no variables are lost in the transformation (i.e., $\forall \phi \in \Phi\colon \mathit{Var}(\phi) \subseteq \mathit{Var}(\theta(\phi))$).

For feature-model analysis, we distinguish five properties that a CNF transformation can fulfill: *equivalence*, *equi-satisfiability*, *equi-assignability*, *equi-countability*, and *quasi-equivalence*. While equivalence, equi-satisfiability, and equi-assignability are known from literature on logic, we propose two new properties that are useful for feature-model analyses, equi-countability and quasi-equivalence, as well as a taxonomy that relates all five properties.

Definition 3.2. A CNF transformation θ fulfills the property:

(1) *Equivalence* [23] if the input formula ϕ and the computed CNF $\theta(\phi)$ are *logically equivalent*:
$$\forall \phi \in \Phi\colon \mathit{Var}(\phi) = \mathit{Var}(\theta(\phi)) \land [\![\phi]\!] = [\![\theta(\phi)]\!]$$
Thus, the resulting CNF shares the same set of variables and the same set of satisfying assignments.

(2) *Equi-satisfiability* [21, 23, 40, 66] if θ preserves *satisfiability* between the input formula and the computed CNF:
$$\forall \phi \in \Phi\colon [\![\phi]\!] = \varnothing \Leftrightarrow [\![\theta(\phi)]\!] = \varnothing$$
Thus, either both formulas are satisfiable or none is.

(3) *Equi-assignability* if θ preserves *satisfying assignments* between the input formula and the computed CNF:
$$\forall \phi \in \Phi\colon [\![\phi]\!] = \{A \cap \mathit{Var}(\phi) \mid A \in [\![\theta(\phi)]\!]\}$$
Thus, each satisfying assignment of the original ϕ can be extended to some (not necessarily unique) satisfying assignment of the transformed $\theta(\phi)$. In logic, equi-assignability is also known as $\mathit{Var}(\phi)$-restricted equivalence [22, 23, 40].

(4) *Equi-countability* if θ preserves the *model count* between the input formula and the computed CNF:
$$\forall \phi \in \Phi\colon |[\![\phi]\!]| = |[\![\theta(\phi)]\!]|$$
Thus, both formulas have the same number of satisfying assignments.

(5) *Quasi-equivalence* if θ fulfills both equi-assignability and equi-countability.

These five properties can be used for classifying CNF transformations and judging their usefulness for different feature-model analyses. To understand how the properties relate to each other, we show which property implies another in Figure 2. An arrow from one property to another means that the former implies the latter. So, equivalence of formulas is the "strongest" property listed, while equi-satisfiability is the "weakest". We prove these claims.

$$\begin{tikzcd}
& \text{Equivalence} \arrow[d] & \\
& \text{Quasi-equivalence}^* \arrow[dl] \arrow[dr] & \\
\text{Equi-assignability} \arrow[dr] & & \text{Equi-countability}^* \arrow[dl] \\
& \text{Equi-satisfiability} &
\end{tikzcd}$$

Figure 2: Taxonomy of properties that a CNF transformation can fulfill (arrows are implications, * are newly proposed).

Theorem 3.3. *The causal relationships depicted in Figure 2 hold.*

Proof. We sketch each relationship's proof individually.

- Equivalence implies quasi-equivalence, which can be seen by substituting $\mathit{Var}(\phi)$ with $\mathit{Var}(\theta(\phi))$ and $[\![\phi]\!]$ with $[\![\theta(\phi)]\!]$ in the conditions of equi-assignability and equi-countability.
- Quasi-equivalence implies both equi-assignability and equi-countability by definition.
- Equi-countability implies equi-satisfiability, because a formula ϕ is satisfiable precisely when $|[\![\phi]\!]| > 0$.
- Equi-assignability implies equi-satisfiability, because both sets in the definition of equi-assignability are empty precisely when $[\![\phi]\!]$ and $[\![\theta(\phi)]\!]$ are. $\square$

The newly proposed quasi-equivalence is particularly useful, because it is less strict than equivalence, but compatible with most feature-model analyses (cf. Section 3.2): While two formulas are only equivalent when their satisfying assignments are completely identical, quasi-equivalence only requires that satisfying assignments are sufficiently similar (i.e., *isomorphic*). This means there is a bijective function that maps a satisfying assignment of one formula to a unique satisfying assignment of the other formula, and vice versa. We prove this claim in the following theorem.

Theorem 3.4. *Let $\phi \in \Phi$ be a formula and θ a CNF transformation that fulfills quasi-equivalence. Then, ϕ and $\theta(\phi)$ have isomorphic satisfying assignments (i.e., there is a bijection between them).*

Proof. Consider the mapping $\sigma\colon A \mapsto A \cap \mathit{Var}(\phi)$. Due to equi-assignability, σ surjects onto $[\![\phi]\!]$. Due to equi-countability, σ is a surjection between sets of the same size, so it is also an injection. Thus, σ is a bijection from $[\![\theta(\phi)]\!]$ to $[\![\phi]\!]$. $\square$

Theorem 3.4 justifies practical usage of quasi-equivalence, as σ allows us to convert between satisfying assignments of the original formula ϕ and the transformed formula $\theta(\phi)$ by ignoring all newly introduced variables in $\theta(\phi)$, if any. Still, quasi-equivalence is weaker than actual equivalence because Theorem 3.4 does not apply to the non-satisfying assignments of $\theta(\phi)$. That is, θ does not generally preserve the model count of the negation $\neg\phi$.

3.2 Requirements for Feature-Model Analysis

In practice, various CNF transformation tools are used to perform feature-model analyses [55, 65, 78, 88, 106, 112]. When using such tools, it is often overlooked whether an analysis is actually compatible with the used transformation. However, this compatibility is vital to ensure that a feature-model analysis returns correct results.

Our taxonomy of CNF transformations (cf. Figure 2) can be used to analyze when a transformed formula $\theta(\phi)$ can be interchangeably used instead of the original formula ϕ, without affecting analysis results.

As an example for this approach, we state the following theorem, which characterizes the requirements of basic feature-model analyses (cf. Section 2.3) regarding the chosen CNF transformation.

Theorem 3.5. *Let $\phi \in \Phi$ be a feature-model formula, θ a CNF transformation, and $f \in Var(\phi)$ a feature. If θ fulfills:*

(1) Equi-satisfiability, then $void(\phi) \Leftrightarrow void(\theta(\phi))$
(2) Equi-assignability, then $dead/core(\phi, f) \Leftrightarrow dead/core(\theta(\phi), f)$
(3) Equi-countability, then $\#fm(\phi) = \#fm(\theta(\phi))$
(4) Quasi-equivalence, then $\#f(\phi, f) = \#f(\theta(\phi), f)$

In Theorem 3.5,[1] we find that to calculate dead and core features, the used CNF transformation must fulfill equi-assignability. In general, equi-assignability is required for all feature-model analyses that gather information about specific features (e.g., for interactive configuration [50, 64], anomaly explanation [60], evolution [61, 116], testing [57, 90], type checking [5, 56], and model checking [6, 97]). If the cardinality of features is also relevant (e.g., for uniform random sampling [83]), quasi-equivalence is required.

If we ignore these requirements and attempt to perform a feature-model analysis with a CNF transformation that *does not* fulfill the required transformation property, the analysis result may be incorrect. Thus, it is the responsibility of researchers and practitioners to carefully consider the requirements of a feature-model analysis and choose a CNF transformation accordingly.

4 CNF TRANSFORMATION ALGORITHMS

In Section 3, we explained how feature-model analyses require a CNF transformation to fulfill certain properties. To apply this knowledge in practice, we describe three well-known CNF transformation algorithms [51, 98] and discuss their suitability for feature-model analysis. We focus on these transformations because they are the foundation on which concrete CNF transformation tools are built [51, 111]. In practice, these transformations are instantiated in different ways (e.g., with optimizations, heuristics, or hybrid transformation schemes [3, 20, 25, 49, 122]), so we focus on the underlying ideas and omit proofs.

4.1 Distributive Transformation

The distributive transformation [23] is a simple approach for transforming any propositional formula into a logically equivalent formula in CNF by applying standard laws of propositional logic [23]. To this end, first all negations $\neg$ can be pushed down towards variables x_i (by De Morgan's laws), which brings the formula into *negation normal form* (NNF). In a second step, all disjunctions $\vee$ can be interchanged with conjunctions $\wedge$ (by distributivity) to create a conjunction of disjunctions of literals and thus a CNF.

[1] Claim (1) and (3) hold by definition of equi-satisfiability and equi-countability. Claim (2) is proven by Schröter et al. [101, Theorem 15]. Claim (4) follows from the same theorem by using the bijection σ—for brevity, we omit a proof.

Definition 4.1. We define the *distributive transformation* θ_D as the negation normal form step *nnf*:

$$nnf(\phi) := \begin{cases} \phi & \phi \text{ is a literal} \\ nnf(\psi) & \phi = \neg\neg\psi \\ nnf(\bigvee_i \neg\psi_i) & \phi = \neg(\bigwedge_i \psi_i) \\ nnf(\bigwedge_i \neg\psi_i) & \phi = \neg(\bigvee_i \psi_i) \\ \bigwedge_i nnf(\psi_i) & \phi = \bigwedge_i \psi_i \\ \bigvee_i nnf(\psi_i) & \phi = \bigvee_i \psi_i \end{cases}$$

followed by the distributive step *dist*:

$$dist(\phi) := \begin{cases} \bigwedge_i dist(\psi_i) & \phi = \bigwedge_i \psi_i \\ \bigwedge_{j_i}^{m_i} (\bigvee_i^n \psi_{i,j_i}) & \phi = \bigvee_i^n \psi_i, dist(\psi_i) = \bigwedge_{j_i}^{m_i} \psi_{i,j_i} \\ \phi & \text{otherwise} \end{cases}$$

So, $\theta_D := dist \circ nnf$ (i.e., transform into NNF first by applying De Morgan's laws, then into CNF by applying distributivity).

To illustrate the distributive transformation on a feature model, recall the example from Figure 1 and its formula ϕ_{GPL} (which is not in CNF). First, we bring ϕ_{GPL} into NNF, then into CNF as follows:

$$\begin{aligned} \phi_{GPL} &= G \wedge E \wedge A \wedge (D \vee U) \wedge (\neg D \vee \neg U) \\ &\quad \wedge (Co \vee Cy) \wedge (\neg(Co \vee Cy) \vee D) \\ nnf(\phi_{GPL}) &= \ldots \wedge (Co \vee Cy) \wedge ((\neg Co \wedge \neg Cy) \vee D) \\ \theta_D(\phi_{GPL}) &= \ldots \wedge (Co \vee Cy) \wedge (\neg Co \vee D) \wedge (\neg Cy \vee D) \end{aligned}$$

Note that we only need to transform the last three conjuncts, as the feature-model formula is a conjunction and all other conjuncts were already (unary) disjunctions of literals. This leverages the first case of the definition of *dist*, by which a conjunction can be equivalently transformed by transforming all conjuncts separately. This is a useful property for analyzing feature models, which can consist of thousands of conjuncts that express constraints on features.

It has been proven [23] that the distributive transformation fulfills equivalence (cf. Section 3.1), so it can safely be used for feature-model analysis (e.g., as done elsewhere [13, 62, 65, 68, 69, 78, 106, 116]). However, it has a serious drawback: The distributive step performs a complex multiplication of clauses, which may lead to an exponential growth of the transformed formula [17, 20, 22, 98, 120]. This issue can be somewhat mitigated by performing additional simplifications using idempotence, absorption, or the Quine–McCluskey algorithm [76, 99]. However, these techniques have limited applicability or exponential time complexity [121].

4.2 Tseitin Transformation

To avoid the exponential explosion of the distributive transformation, Tseitin [120] proposed a different approach for transforming formulas into CNF. The Tseitin transformation *replaces* each subformula ϕ of a non-CNF formula with a new Boolean variable x_ϕ, which acts as a "shortcut" to refer to ϕ. The new variable x_ϕ is then tied to the original subformula ϕ using a biimplication $\leftrightarrow$. Besides addressing the exponential explosion of the distributive transformation, the Tseitin transformation aims to remove identical subformulas in the formula, which may result in a more concise formula [22].

Because the original description is informal and thus vague, it is difficult to give a standard definition of the Tseitin transformation.

We propose a succinct definition along the lines of Bradley and Manna [21] to illustrate the procedure for binary $\land$ and $\lor$.

Definition 4.2. We write $\psi \sqsubseteq \phi$ to denote that ψ is a subformula of ϕ. The *Tseitin transformation* θ_T is then given by the NNF step *nnf* followed by the replacement step $rep_{\leftrightarrow}$ given by

$$rep_{\leftrightarrow}(\phi) := \langle\phi\rangle \land \bigwedge_{\substack{(\psi * \chi)\sqsubseteq\phi \\ *\in\{\land,\lor\}}} clausify(\langle\psi * \chi\rangle \leftrightarrow (\langle\psi\rangle * \langle\chi\rangle))$$

$$\text{where } \langle\phi\rangle := \begin{cases} \phi & \phi \text{ is a literal} \\ x_\phi & \text{otherwise } (x_\phi \text{ being an auxiliary variable}) \end{cases}$$

and *clausify* looks up a precomputed CNF expression.[2] So, $\theta_T := rep_{\leftrightarrow} \circ nnf$ (i.e., transform into NNF first, then into CNF by introducing auxiliary variables x_ϕ that are tied to subformulas ϕ).

To illustrate the Tseitin transformation, we use it to transform the feature-model formula ϕ_{GPL} into CNF. As with the distributive transformation, we begin with the NNF $nnf(\phi_{GPL})$ and then apply $rep_{\leftrightarrow}$, arriving at the transformed formula $\theta_T(\phi_{GPL})$:

$$\begin{aligned}\theta_T(\phi_{GPL}) = x_1 & \\ &\land clausify(x_1 \leftrightarrow (G \land E \land A \land x_2 \land x_3 \land x_4 \land x_5)) \\ &\land clausify(x_2 \leftrightarrow (D \lor U)) \land clausify(x_3 \leftrightarrow (\neg D \lor \neg U)) \\ &\land clausify(x_4 \leftrightarrow (Co \lor Cy)) \land clausify(x_5 \leftrightarrow (x_6 \lor D)) \\ &\land clausify(x_6 \leftrightarrow (\neg Co \land \neg Cy))\end{aligned}$$

In $\theta_T(\phi_{GPL})$, we represent each non-trivial subformula $\psi \sqsubseteq \phi_{GPL}$ by assigning it an auxiliary variable $\langle\psi\rangle = x_i$ as a shortcut (for brevity, we index these variables with integers instead of formulas). This way, we flatten the formula into a conjunction of six shortcut definitions $x_1, \ldots, x_6$. For improved readability, we write these definitions with $\leftrightarrow$ inside *clausify*, but in practice, they can be directly generated in CNF. Finally, to "activate" the entire formula $\theta_T(\phi_{GPL})$, we add its root auxiliary variable $\langle\phi_{GPL}\rangle = x_1$ as a unit clause.

As this example illustrates, the Tseitin transformation θ_T introduces new variables and may therefore result in formulas larger than necessary. However, contrary to θ_D, the Tseitin transformation θ_T is guaranteed to be asymptotically linear in time and space (in size of the original formula) because of the flattened formula structure [20, 98, 120].

With regard to our taxonomy, the Tseitin transformation θ_T does not fulfill equivalence (cf. Section 3.1), as it introduces new variables. However, it has been proven that θ_T fulfills both equi-assignability [93] and equi-countability [104]. So, θ_T fulfills quasi-equivalence, which can be related to equivalence by ignoring all auxiliary variables from satisfying assignments after calling a SAT solver (cf. Theorem 3.4). Thus, the Tseitin transformation can be safely used for all feature-model analyses discussed in Section 3.2 (e.g., as done elsewhere [13, 88, 91, 112]).

4.3 Plaisted-Greenbaum Transformation

Two decades after Tseitin, Plaisted and Greenbaum [93] proposed a variant of the Tseitin transformation, which we refer to as the Plaisted-Greenbaum transformation (also called *polarity-based encoding* [22, 52, 98]). They noticed that many clauses added by the Tseitin transformation can be omitted by taking the *polarity* of each subformula into account (i.e., the parity of negations in the syntax tree between the root and the subformula). Then, instead of defining auxiliary variables with biimplications $\leftrightarrow$, it suffices to use $\rightarrow$ (for positive polarity) and $\leftarrow$ (for negative polarity) [17, 22, 47, 51, 93].

When a formula is transformed into NNF first, all subformulas have positive polarity and we can succinctly define the Plaisted-Greenbaum transformation as follows.

Definition 4.3. The *Plaisted-Greenbaum transformation* is given by $\theta_{PG} := rep_{\rightarrow} \circ nnf$ (i.e., transform into NNF first, then into CNF by introducing variables x_ϕ that imply some subformula ϕ).

It has been proven [93] that the Plaisted-Greenbaum transformation θ_{PG} introduces fewer clauses than the Tseitin transformation θ_T and still fulfills equi-assignability, so it can be safely used for some feature-model analyses (e.g., as done elsewhere [55, 56, 84]). However, θ_{PG} also has a disadvantage compared to θ_T: By using $\rightarrow$ instead of $\leftrightarrow$, θ_{PG} only encodes the satisfiability of subformulas, so the number of satisfying assignments is not preserved by the transformation. Thus, θ_{PG} fulfills neither equi-countability [67, 88] nor quasi-equivalence. This makes the Plaisted-Greenbaum transformation unsuitable for cardinality analyses (cf. Section 2.3), which rely on the configuration space being free of distortions.

5 EVALUATION

In this section, we complement our theoretical discussion in Section 3 and 4 by evaluating the practical impact of CNF transformations on the performance and correctness of feature-model analyses.

5.1 Research Questions

In our evaluation, we aim to determine whether the choice of a CNF transformation tool has a noticeable impact on the work of practitioners and researchers. We consider a tool's impact noticeable if it (a) fails to create CNFs in reasonable time, (b) significantly affects the runtime of analyses, or (c) subsequently causes incorrect analysis results. Thus, we pose the following research questions:

RQ$_1$ How efficient are CNF transformation tools?

RQ$_2$ Does the efficiency of subsequent feature-model analyses depend on the CNF transformation tool?

RQ$_3$ Does every CNF transformation tool yield correct analysis results?

We answer these questions by running three representative CNF transformation tools on a corpus of real-world feature models and performing feature-model analyses on the transformed formulas. Thus, we aim to realistically reproduce the impact that CNF transformations have on feature-model analyses.

5.2 Subject Systems and Experimental Setup

To answer these research questions, we set up an evaluation in three subsequent stages, as depicted in Figure 3. We (1) *extract* a propositional *formula* from the feature model of a given product line, (2) *transform* the formula into *CNF*, and (3) perform *automated analyses* on the CNF. In Stage 2, we set the timeout for executing transformations to three minutes; analogously, we set the timeout for executing solvers in Stage 3 to 20 minutes. We repeat each measurement three times and analyze median values to mitigate

[2]For instance, $clausify(a \leftrightarrow (b \land c)) = (\neg a \lor b) \land (\neg a \lor c) \land (a \lor \neg b \lor \neg c)$.

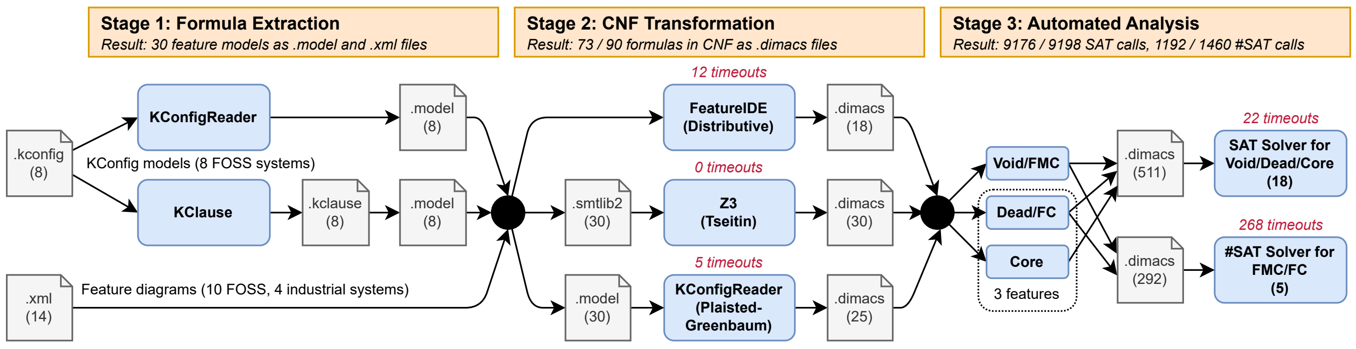
Stage 1: Formula Extraction
Result: 30 feature models as .model and .xml files
Stage 2: CNF Transformation
Result: 73 / 90 formulas in CNF as .dimacs files
Stage 3: Automated Analysis
Result: 9176 / 9198 SAT calls, 1192 / 1460 #SAT calls
.kconfig (8)
KConfigReader
KClause
KConfig models (8 FOSS systems)
.model (8)
.kclause (8)
.model (8)
.xml (14)
Feature diagrams (10 FOSS, 4 industrial systems)
12 timeouts
FeatureIDE (Distributive)
0 timeouts
.smtlib2 (30)
Z3 (Tseitin)
5 timeouts
.model (30)
KConfigReader (Plaisted-Greenbaum)
.dimacs (18)
.dimacs (30)
.dimacs (25)
Void/FMC
Dead/FC
Core
3 features
.dimacs (511)
.dimacs (292)
22 timeouts
SAT Solver for Void/Dead/Core (18)
268 timeouts
#SAT Solver for FMC/FC (5)

Figure 3: Evaluation stages for comparing three tools that transform real-world feature models into CNF.

Table 1: Systems and their feature-model formulas, ordered by number of variables (read by KConfigReader/KClause).

Source	System	Version	#Var	#Lit
KConfig (extracted)	Fiasco	58aa50a	85/116	554/856
	axTLS	2.0.0	123/185	640/1227
	uClibc-ng	1.0.40	246/388	3874/8421
	BusyBox	1.35.0	866/986	5159/4728
	EmbToolkit	1.8.0	2510/4231	28k/108k
	Buildroot	2021.11.2	7548/9228	215k/584k
	Freetz-NG	5c5a4d1	9k/11k	1119k/2713k
	Linux (x86)	4.18	13k/22k	223k/1072k
KConfig (hierarchy)	axTLS	N/A	96	414
	uClibc	N/A	313	2739
	uClinux-base	N/A	380	15k
	BusyBox	1.18.0	854	2666
	EmbToolkit	N/A	1179	35k
	uClinux-dist	N/A	1580	4201
	Linux (x86)	2.6.33.3	6467	45k
CDL (3/116)	Min: am31_sim	N/A	1178	5669
	Median: lpcmt	N/A	1262	6167
	Max: ea2468	N/A	1408	6908
Closed-Source	Automotive (4)	2.1-2.4	14k-19k	485k-714k

measurement inaccuracies. We choose timeouts of three and 20 minutes as well as three repetitions because this appears to be a reasonable tradeoff between invested computational resources (e.g., environmental impact) and robustness of the results. We run the evaluation on an Intel Xeon E5-2630 PC with 2.40GHz and 128GiB RAM. For reproducibility, we disclose our fully automated, Docker-based evaluation pipeline[3] and all feature models, solvers, and results in form of a replication package.[4]

5.2.1 Formula Extraction. In Stage 1, we prepare a corpus of formulas to be transformed and analyzed in Stage 2 and 3 by extracting them from feature models of 22 systems. Our corpus is based on two complementary sources: recent KConfig models (extracted by ourselves) and feature diagrams (extracted by Knüppel et al. [59]).

KConfig Models KConfig[5] is a textual configuration language mostly used in systems programming and embedded software [35]. It was originally developed for managing variability in the Linux kernel, but has been adapted by several projects in the free and open-source software (FOSS) community (cf. Table 1).

For our evaluation, we extract formulas for recent versions of eight FOSS systems, which we list in Table 1 as *KConfig (extracted)*. We found older versions of these systems in a study of twelve FOSS systems by Berger et al. [13], from which we exclude the four systems Coreboot, Toybox, uClinux-base, and uClinux-dist, because we were unable to extract their feature-model formulas due to custom KConfig parsers and missing vendor files. For the eight listed systems, we extract two feature-model formulas from the system's KConfig files using the tools KConfigReader and KClause, respectively. The tool KConfigReader[6] [55] is part of the open-source infrastructure TypeChef [56] for analyzing configurable systems written in C. According to El-Sharkawy et al. [35], KConfigReader results in more accurate translations than comparable tools (i.e., LVAT [106], Undertaker [112]). Still, we also extract formulas with KClause[7] [88] to reduce the risk of our results being only evident for a certain extraction method. Both tools produce non-CNF formulas, and thus are suitable for evaluating the transformations.

Knüppel's Models In addition to the KConfig models, we also consider feature diagrams (cf. Figure 1), which are a well-known hierarchical representation of feature models [54]. We use feature diagrams in FeatureIDE XML format extracted by Knüppel et al. [59], which have been used in several publications on feature-model analysis [63, 64, 94, 110]. Knüppel et al. extracted seven feature models from KConfig using LVAT [106] and 116 feature models from CDL files in eCos [113] using CDLTools [14]. In addition, they include four feature models that have been contributed by an industrial partner in the automotive domain. Regarding eCos, we only consider three of their 116 extracted CDL models, namely those with a minimum, median, and maximum number of features, as theses models are highly homogeneous [59] and we want to prevent an overrepresentation of those CDL models.

[3]Automation scripts available at: https://doi.org/10.5281/zenodo.6922807
[4]Replication package available at: https://doi.org/10.5281/zenodo.6525375

[5]https://github.com/torvalds/linux/blob/master/Documentation/kbuild/kconfig-language.rst
[6]https://github.com/ckaestne/kconfigreader
[7]https://github.com/paulgazz/kmax

5.2.2 *CNF Transformation.*

In Stage 2, we transform the feature-model formulas from Stage 1 into CNF. For the transformation, we consider three tools: FeatureIDE, Z3, and KConfigReader. Each tool implements some variant of a CNF transformation discussed in Section 4 and is used in practice for feature-model analysis [13, 56, 78, 88, 91, 116]. For each of the three tools, we translate each feature-model formula into a suitable input format (i.e., XML, SMT-LIB 2 [7], or `.model`) and receive a DIMACS file containing the corresponding CNF [33]. We measure the time required for the entire transformation process of each tool, including I/O effort.

FeatureIDE FeatureIDE [78, 117] is an open-source IDE for modeling, implementing, and testing configurable software systems. It is widely used to create and analyze real-world feature models [59]. For the automated analysis of feature models, FeatureIDE implements the distributive transformation θ_D with some optimizations (e.g., subsumption of clauses and optional unit propagation).[8] Thus, we expect it to fulfill equivalence (cf. Section 3.1) and thus produce correct results for all feature-model analyses in Section 3.2.

Z3 The open-source SMT solver Z3 [32] implements the `tseitin-cnf` tactic, which is a hybrid variant of the Tseitin transformation θ_T. That is, `tseitin-cnf` introduces an auxiliary variable for a subformula $\psi \sqsubseteq \phi$ if ψ occurs several times in ϕ or the exponential blowup by the distributive transformation would be unacceptable—otherwise, it uses the distributive transformation on ψ. In our evaluation, we use the default threshold for determining the acceptable blowup. In addition, `tseitin-cnf` implements several optimizations (e.g., recognition of common patterns, handling *if-then-else* chains, and precomputed CNF expressions) and skips subformulas that are already in CNF. As Z3's `tseitin-cnf` tactic uses biimplications $\leftrightarrow$ to replace subformulas[9] [83], we expect it to fulfill quasi-equivalence, and thus also produce correct results for the feature-model analyses in Section 3.2.

KConfigReader In addition to extracting feature-model formulas, KConfigReader can also transform formulas into CNF. This transformation step is an optimized implementation of the Plaisted-Greenbaum transformation θ_{PG} [88]. Similar to Z3, KConfigReader only introduces an auxiliary variable for a subformula $\phi \vee \psi$ if the predicted number of clauses for the distributive transformation $|\phi| * |\psi|$ exceeds a fixed threshold.[10] In contrast to Z3, KConfigReader uses implications $\rightarrow$ for replacing subformulas. Thus, we expect it to only fulfill equi-assignability and produce potentially incorrect results for cardinality analyses.

5.2.3 *Automated Analysis.*

In Stage 3, we perform several feature-model analyses on the CNFs originating from Stage 2 by constructing corresponding DIMACS files and passing them to the SAT and #SAT solvers listed in Table 2. For each CNF ϕ, we construct DIMACS files, containing the following:

(1) ϕ — for checking whether the feature model is *void* (SAT) and calculating the *feature-model cardinality (FMC)* (#SAT).

[8]https://github.com/FeatureIDE/FeatureIDE/blob/b3722d/plugins/de.ovgu.featureide.fm.core/src/org/prop4j/Node.java

[9]As evidenced by adding clauses $\{\neg k, l_1, \dots, l_n\}$ for $\rightarrow$ and $\{\neg l_i, k\}$ for $\leftarrow$: https://github.com/Z3Prover/z3/blob/fc7734/src/tactic/core/tseitin_cnf_tactic.cpp#L715

[10]Clause number prediction in l. 814 and adding clauses $\{\neg k, l_1, \dots, l_n\}$ for $\rightarrow$ in l. 875: https://github.com/ckaestne/TypeChef/blob/fb4d7a/FeatureExprLib/src/main/scala/de/fosd/typechef/featureexpr/sat/SATFeatureExpr.scala#L799

Table 2: Evaluated SAT and #SAT solvers.

*Winner in SAT Competition †Participant (‡Winner) in MC Competition

Class	Year	Solver	Class	Year	Solver
SAT	2002*, 04*	zChaff	SAT	2013–14*	Lingeling
	2003*	Forklift		2016–19*	MapleSAT
	2005*	SatELiteGTI		2020–21*	Kissat
	2006*	MiniSat	#SAT	-	countAntom
	2007*	RSat		2020†	d4
	2009*	PrecoSAT		-	dSharp
	2010*	CryptoMiniSat		2020‡	Ganak
	2011–12*	Glucose		2021‡	sharpSAT

(2) $\phi \wedge f$ with a randomly chosen feature f — for checking whether f is *dead* (SAT) and calculating the *feature cardinality (FC)* of f (#SAT).

(3) $\phi \wedge \neg f$ with a randomly chosen feature f — for checking whether f is *core* (SAT).

For File (2) and (3), we choose three different random features, but choose the same three features for each transformation tool to ensure a fair comparison. We do not call a #SAT solver for File (3), as this number can be calculated simply by subtracting the FC of f (2) from the FMC (1). We pass these DIMACS files to multiple solvers and measure their respective runtimes.

For selecting solvers, we aim to include solvers which perform well or are used in feature-model analysis. First, we evaluate all publicly available SAT solvers that won a gold medal in the main track of any SAT competition[11] [53]. Second, we evaluate the five fastest #SAT solvers to-date for feature-model analysis identified by Sundermann et al. [110], which include the best-performing #SAT solvers in the model-counting competition 2020 [38] and 2021.[12]

5.3 Results and Discussion

In the following, we describe and discuss the evaluation results for our three research questions.

In Figure 4, we visualize our time measurements for the CNF transformation (**RQ$_1$**) and the performed feature-model analyses (**RQ$_2$**). On the x-axis, we show the evaluated algorithms (i.e., the transformation and analyses). Each colored box contains the data for one transformation tool. On the logarithmic y-axis, we show the time required by each tool. To avoid a comparison of absolute values, which may lead to false representation of large feature models, all values on the y-axis are relative to the variant of the Tseitin transformation implemented in Z3. Consequently, all results for Z3 are on the dashed line at y-value 1, and a value $y > 1$ is to be read as "y times slower than Z3" (or "$1/y$ times faster than Z3" for $y < 1$). We exclude measurements when a transformation tool either reached the timeout of three minutes (20 minutes for solvers, respectively) or when it crashed without a result (e.g., due to limited heap space). Similarly, we exclude outliers above factor 100 to improve the visualization. We indicate the percentage of *included* measurements as a percentage below and above each box: For example, a 60% below a box means there are 40% N/A values, and a 100% above a box means there are no outliers.

[11]http://www.satcompetition.org/

[12]https://mccompetition.org/

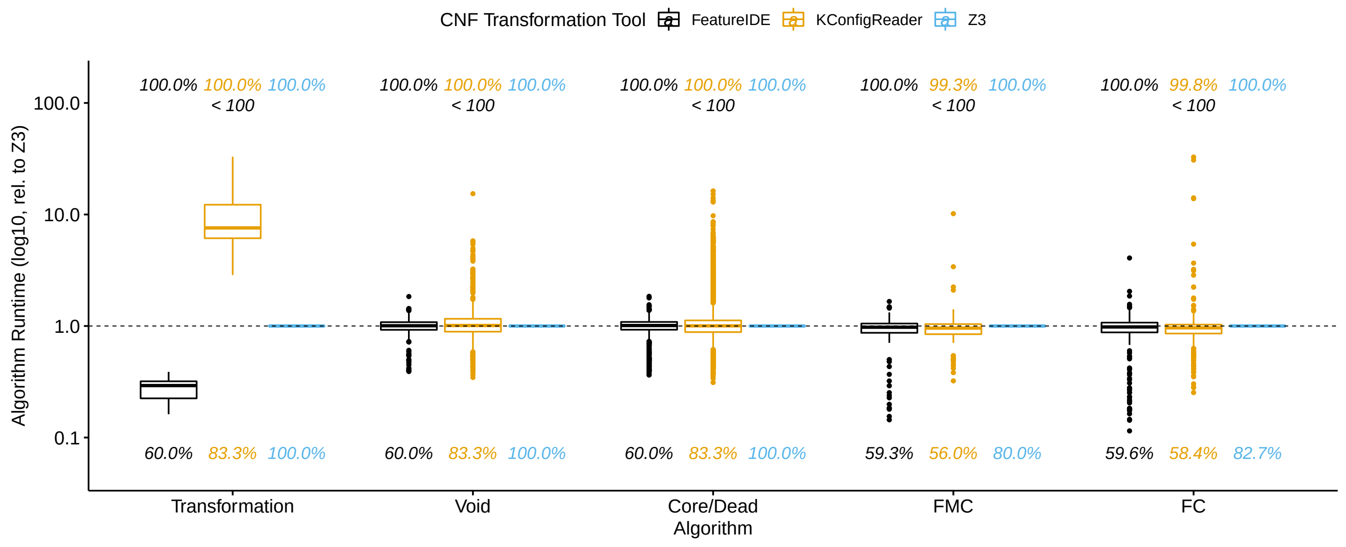

Figure 4: Impact of CNF transformation tool on runtime relative to Z3 (RQ$_1$: transformation runtime, RQ$_2$: analysis runtime).

Table 3: Results of paired t-test on absolute runtime.

RQ	Compared CNF Transformations		p-Value	Effect Size
RQ$_1$	FeatureIDE	KConfigReader	8.73×10^{-8}	-3.12
	FeatureIDE	Z3	2.00×10^{-3}	-9.34×10^{-1}
	KConfigReader	Z3	1.60×10^{-2}	5.20×10^{-1}
RQ$_2$	FeatureIDE	KConfigReader	4.20×10^{-2}	-5.44×10^{-2}
	FeatureIDE	Z3	4.20×10^{-2}	-4.50×10^{-2}
	KConfigReader	Z3	8.60×10^{-1}	-3.01×10^{-3}

Table 4: Quartiles of analysis runtime relative to Z3 (RQ$_2$).

Solver	Transformation	Min	Q1	Median	Q3	Max
SAT	FeatureIDE	0.36	0.93	1.01	1.09	1.85
	KConfigReader	0.31	0.88	1.00	1.13	16.27
#SAT	FeatureIDE	0.11	0.88	0.98	1.07	4.07
	KConfigReader	0.25	0.85	0.96	1.03	3114.55

For **RQ$_1$** and **RQ$_2$**, we apply paired t-tests on our measurements to reason whether the choice of a transformation tool has a significant ($p < 0.05$) impact on transformation and analyses time. We report the p-values and effect sizes of these tests in Table 3.

5.3.1 RQ$_1$: Impact on Transformation Runtime. To determine the efficiency of each considered CNF transformation tool, we measure its transformation time. In Figure 4, we show the relative transformation times for each tool for the 30 feature models from Stage 1.

Results We observe that FeatureIDE, KConfigReader, and Z3 succeed in transforming 60%, 83%, and 100% of feature-model formulas within three minutes, respectively. For the 60% of models that were successfully transformed by all tools, FeatureIDE, KConfigReader, and Z3 required a median transformation time of 66 ms, 1359 ms, and 223 ms, respectively. Overall, in the cases where FeatureIDE succeeds, it is significantly ($p = 0.002$) faster than Z3, while KConfigReader is significantly ($p = 0.016$) slower than Z3.

Discussion FeatureIDE fails to transform 40% of feature models into CNF, while KConfigReader fails for 17% and Z3 succeeds on all feature models. Thus, the chosen CNF transformation tool has a significant impact on the ability to create a CNF. The failures of FeatureIDE are likely due to its use of the distributive transformation, which causes an exponential growth when transforming specific types of formulas (cf. Section 4). We suspect that the failures of KConfigReader are due to its lack of optimization compared to the more mature Z3. Nevertheless, the actual performance of a transformation tool heavily depends on the concrete input. For example, FeatureIDE is significantly faster than Z3 for the 60% successfully transformed feature models, while failing for the remaining 40%.

5.3.2 RQ$_2$: Impact on Analysis Runtime. To determine the impact of the CNF transformation on feature-model analyses, we measure the time required to execute a solver query. In Figure 4, we show the execution times for all solver calls from Stage 3 performed on the successfully transformed CNFs in Stage 2 by each tool and relative to Z3. The percentage values below boxes show the ratio of successful solver calls, including the transformation failures from Stage 2. For example, we were able to perform the void analysis for all CNFs computed by FeatureIDE (i.e., 60%). In Table 4, we report the quartiles of the boxes from Figure 4, separated by solver class.

Results First, we observe that not all solver calls were successful. Of the 9,198 attempted SAT calls, 22 (all for the SAT solver Forklift) did not successfully return a result within 20 minutes. Of the 1,460 attempted #SAT calls, 268 did not successfully return a result within 20 minutes. Second, we see that the solving time for successful solver calls can vary depending on the chosen transformation. When we transform with FeatureIDE, the solving time varies by a factor of 0.11–4.07 compared to the Z3 transformation (cf. Table 4).

Table 5: Share of correct analysis results (RQ$_3$).

Analysis/CNF Trans.	FeatureIDE	Z3	KConfigReader
Void	100%	100%	100%
Dead/Core	100%	100%	100%
FMC	100%	100%	29.412%
FC	100%	100%	29.502%

However, for half of all solver calls, the solving time only varies by a factor of 0.92–1.08. When we transform with KConfigReader, the solving time varies by factor 0.25–3114.55 (or 32.7, excluding the outlier uClibc-ng) compared to Z3. Still, for half of all solver calls, the solving time only varies by factor 0.88–1.11. Transforming with either FeatureIDE, KConfigReader, or Z3 leads to the fastest solving times for 21.64%, 25.4%, or 52.96% of all successful solver calls, respectively. The solving times for the successful calls differ significantly for FeatureIDE and KConfigReader ($p = 0.042$) as well as FeatureIDE and Z3 ($p = 0.042$), but not for KConfigReader and Z3 ($p = 0.86$); the corresponding effect sizes are negligible.

Discussion Our results indicate that the selection of a CNF transformation tool is relevant for the runtime of SAT- and #SAT-based analyses using the transformed CNFs as input. That is, solvers can get up to one order of magnitude faster or slower (or even three orders of magnitude, considering the outlier uClibc-ng) just by switching the CNF transformation. This showcases that the transformation to CNF needs to be considered when (1) optimizing performance in practice and (2) evaluating tools working on CNFs.

5.3.3 RQ$_3$: Impact on Correctness of Analyses. To determine the impact of a CNF transformation tool on the correctness of analysis results, we compare the output of SAT (i.e., *satisfiable* or *unsatisfiable*) and #SAT solver calls (i.e., the reported model count) for the same analysis on the same feature model. In Table 5, we show the percentage of successful solver calls leading to correct results for the three CNF transformation tools. The percentages are in relation to all solver calls that successfully returned some result. As we have no ground truth for judging correctness, we consider a result correct if it coincides with the result obtained for at least one other CNF transformation tool. For instance, 100% for void analysis with FeatureIDE means that for each solver call, at least one other tool (in this case, both KConfigReader and Z3) led to the same result.

Results For each SAT query (i.e., void analysis and dead/core feature), each SAT solver computed the same result independent of the CNF transformation that was used. For #SAT, the cardinalities of feature models and features were equal for all successfully transformed feature models and every #SAT solver when using Z3 and FeatureIDE (excluding a known solver bug in dSharp [39]). When using CNFs transformed by KConfigReader as input, only 29.412% of the results for feature-model cardinality (FMC) and 29.502% for feature cardinality (FC) are equal to the results computed with the other two transformations. In Figure 5, we show how much the incorrect results produced by KConfigReader-transformed CNFs deviate from the correct ones for cardinality analyses. On the x-axis, we show the evaluated CNF transformation tools; on the logarithmic y-axis, we show the model count reported by the

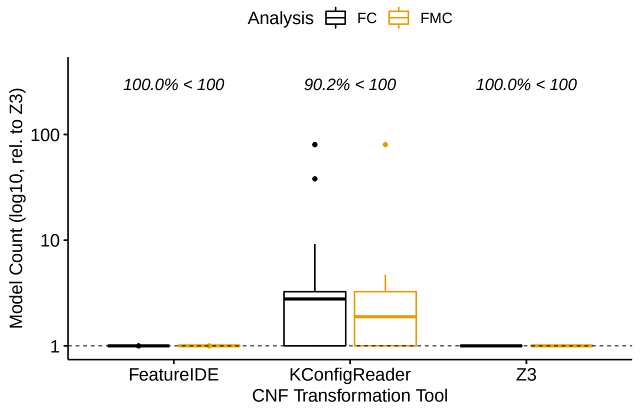

Figure 5: Deviation from correct analysis results (RQ$_3$).

#SAT solvers. Similar to Figure 4, the y-axis is relative to Z3 and therefore measures how incorrect the results are (to improve the visualization, we exclude outliers above factor 100). For the feature-model cardinality, the incorrect results are 3.14 times as large as the correct one in the median and 1.063×10^{77} times as large in the worst case. For the feature cardinality, the incorrect results are 3.26 times as large in the median and 1.345×10^{77} times as large in the worst case.

Discussion When selecting a CNF transformation without considering its properties and limits, there is a considerable risk of not only worsening performance but also to acquire wrong results. In our evaluation, using KConfigReader (which implements the Plaisted-Greenbaum transformation) leads to wrong cardinalities in many cases, matching the results of Oh et al. [88]. Thus, our insights on the correctness of analysis results match the expectations of our theoretical analysis of CNF transformations in Section 4: On the one hand, all three equi-assignable CNF transformation tools, namely FeatureIDE (an instance of θ_D), Z3 (an instance of θ_T), and KConfigReader (an instance of θ_{PG}) lead to correct results with SAT solvers. On the other hand, only the two quasi-equivalent CNF transformation tools, namely FeatureIDE and Z3, always computed correct results with #SAT solvers. We argue that the match between the behavior of the actual implementations and our conceptual analyses validates our concept and indicates the correctness of the implementations.

5.4 Threats to Validity

In the following, we list several threats to the validity of our evaluation, distinguishing internal and external validity [123].

5.4.1 Internal Validity. It is possible that a measurement varies in runtime due to several factors impacting the performance (e.g., warm-up effects or randomness). To reduce such impacts, we repeat each measurement three times and use the median in our results. Also, we consider several analyses and feature models, and thus data points, for each solver/transformation combination, repeated our evaluation on a different machine (an AMD Ryzen 9 5900HX PC with 3.30GHz und 32GiB RAM) with similar results, and disclose all our results publicly. In addition, we employ statistical tests to show the significance of our results. To verify the correctness of the evaluated tools, we manually checked the plausibility of our results (e.g., by comparing cardinalities). In particular, the #SAT solver Ganak is probabilistic and therefore calculates results only with a given confidence; however, it usually returns correct results [105], which we manually confirmed for our evaluation.

5.4.2 External Validity. First, our conclusions cannot be necessarily translated to other feature models. However, to allow representative insights, we evaluated feature models of varying sizes (85-22k variables) and sources (extracted models and hierarchies). Second, we perform a practical evaluation of concrete tools for CNF transformation and solving. Thus, we cannot make definite claims about the underlying algorithms, as any reported differences may also be due to specific implementation choices instead (e.g., using a parameter to trade off the distributive against the Tseitin transformation). However, we argue that using default parameters of CNF transformations reasonably reflects usage in practice and we also consider parameterizing each transformation as out of scope for this work. Third, our results for solving time may be subject to the influence of preprocessing techniques [15] implemented in modern solvers. For example, blocked clause elimination [51] is likely to undo part of the Tseitin transformation internally [88]. To mitigate such influences, we evaluated the transformations with a large variety of solvers that perform preprocessing to various degrees (e.g., solvers published before 2010 do not eliminate blocked clauses).

6 RELATED WORK

While it is well-recognized in automated reasoning that CNF encoding techniques can impact solver efficiency [17, 51, 98, 107], we are not aware of any study that systematically investigates CNF transformations. Sutcliffe and Melville [111] study variants of the distributive transformation, but they dismiss other transformations. Järvisalo et al. [51] compare the Tseitin and Plaisted-Greenbaum transformations in the context of preprocessing techniques, but do not evaluate the transformations' impact on solvers.

In the feature-modeling community, several translations of feature models into formulas are known [4, 8, 83, 100], but CNF transformations are rarely discussed explicitly. We are only aware of one study by Oh et al. [88], in which they find that some CNF transformations may cause incorrect results for feature-model analysis. In comparison, we systematically investigate transformation properties as well as CNF transformation algorithms, and we contribute the first empirical comparison of CNF transformation tools.

While CNF transformations are rarely discussed in the context of feature modeling, a related operation is known as feature-model slicing [1, 62]. Slices are excerpts of feature models that do not introduce, but remove variables—thus, slicing is dual to variable-introducing CNF transformations (e.g., the Tseitin transformation). In our taxonomy, slicing is an equi-assignable feature-model transformation, so insights about slicing [29, 73, 102] also apply to CNF transformations and vice versa. In addition, Knüppel et al. [59] describe a variable-introducing feature-model transformation that adds abstract features to eliminate complex constraints; this transformation can be regarded as a kind of Tseitin transformation.

Another way to deal with CNF transformations is to avoid them entirely. For example, one can use SMT solvers [32] or non-CNF solvers [42, 70, 81] instead. Alternatively, solvers can be avoided completely by mapping feature models onto knowledge compilation data structures such as BDDs [46]. Nevertheless, translating formulas to CNF and invoking a SAT or #SAT solver still appears to be the most common approach for product-line analyses [9, 115].

Many papers that do rely on a transformation of feature-model formulas into CNF overlook this step [35, 62, 68, 69, 103, 108], discuss it only superficially [8, 9, 37, 45, 64, 71, 96, 116], or mix up transformations [22, 40, 92, 114]. This lack of precision can be traced back to Tseitin's original paper [120], in which the transformation θ_T is only defined informally. Thus, authors disagree whether θ_T should replace *all* subformulas with auxiliary variables [17, 20–22, 47, 51, 77, 93, 98, 120] or not [20, 93, 98] and whether it should replace identical subformulas with the same auxiliary variable or not [20, 85, 87]. Due to these disagreements, several advanced CNF transformations emerged [3, 20, 25, 49, 75, 87, 93, 122]. We find that there is a mismatch between this variety of CNF transformation algorithms and CNF transformation tools, which may explain why stating the precise transformation algorithm is so difficult.

7 CONCLUSION

To develop variant-rich software, many automated analyses in all phases of the software development process make it necessary to transform feature-model formulas into CNF. Nevertheless, this step is rarely taken into serious consideration by researchers and practitioners and is either overlooked or only mentioned superficially in literature. We found that the selection of a CNF transformation has a substantial impact not only on the performance of the transformation itself, but also on the efficiency and even the correctness of subsequent analyses. In particular, there seems to be a tradeoff between using FeatureIDE, a distributive transformation tool that is fast but fails to transform many feature models, and Z3, a Tseitin transformation tool that can transform all models but requires more time. Moreover, for the experiment design at hand, we found that the Plaisted-Greenbaum transformation implemented in KConfigReader has no benefits over the other CNF transformation tools, as its results for cardinality-based analyses are often incorrect and the transformation is less efficient.

In summary, both our theoretical analysis and empirical evaluation show that the selection of CNF transformations is highly relevant for practitioners and researchers, especially when using performance-critical analyses, and has to be considered carefully.

In future work, we aim to evaluate selection criteria to identify the most suitable CNF transformation for a given formula. To this end, we want to investigate further what makes a constraint difficult to transform. We also aim to apply our taxonomy of transformation properties on other feature-model transformations, such as slicing.

ACKNOWLEDGMENTS

We thank Tobias Heß and Paul Maximilian Bittner for their feedback on our draft and Paul Gazzillo, Alexander Kittelmann, and Tobias Pett for their comments on the extraction of feature-model formulas. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. 2011. Slicing Feature Models. In *ASE*. IEEE, 424–427.

[2] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. 2012. Feature Model Differences. In *CAiSE*. Springer, 629–645.

[3] S. Alouneh, S. Abed, M. H. Al Shayeji, and R. Mesleh. 2019. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. *Artificial Intelligence Review* 52, 4 (2019), 2575–2601.

[4] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[5] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. 2010. Type Safety for Feature-Oriented Product Lines. *AUSE* 17, 3 (2010), 251–300.

[6] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. 2011. Detection of Feature Interactions Using Feature-Aware Verification. In *ASE*. IEEE, 372–375.

[7] C. Barrett, P. Fontaine, and C. Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa.

[8] D. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC*. Springer, 7–20.

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

[10] T. Berger and P. Collet. 2019. Usage Scenarios for a Common Feature Modeling Language. In *SPLC*. ACM, 174–181.

[11] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS*. Springer, 302–319.

[12] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM, 7:1–7:8.

[13] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *TSE* 39, 12 (2013), 1611–1640.

[14] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. 2010. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *ASE*. ACM, 73–82.

[15] A. Biere, M. Järvisalo, and B. Kiesl. 2021. Preprocessing in SAT solving. *Handbook of Satisfiability* 336 (2021), 391.

[16] P. M. Bittner, C. Tinnes, A. Schultheiß, S. Viegener, T. Kehrer, and T. Thüm. 2022. Classifying Edits to Variability in Source Code. In *ESEC/FSE*. ACM. To appear.

[17] M. Björk. 2011. Successful SAT encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4 (2011), 189–201.

[18] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI*. ACM, 355–364.

[19] B. Boehm, A. W. Brown, R. Madachy, and Y. Yang. 2004. A Software Product Line Life Cycle Cost Estimation Model. In *ISESE*. IEEE, 156–164.

[20] T. Boy de la Tour. 1992. An optimality result for clause form translation. *Journal of Symbolic Computation* 14, 4 (1992), 283–301.

[21] A. R. Bradley and Z. Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Propositional Logic, 3–34.

[22] U. Bubeck and H. K. Büning. 2010. The power of auxiliary variables for propositional and quantified boolean formulas. *Studies in Logic* 3, 3 (2010), 1–23.

[23] H. K. Büning and T. Lettmann. 1999. *Propositional logic: deduction and algorithms*. Vol. 48. Cambridge University Press.

[24] J. Burchard, T. Schubert, and B. Becker. 2015. Laissez-Faire Caching for Parallel# SAT Solving. In *SAT*. Springer, 46–61.

[25] B. Chambers, P. Manolios, and D. Vroon. 2009. Faster SAT solving with better CNF generation. In *2009 Design, Automation Test in Europe Conference Exhibition*. 1590–1595.

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

[27] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*. ACM, 173–182.

[28] K. Czarnecki and A. Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *SPLC*. IEEE, 23–34.

[29] F. Damiani, M. Lienhardt, and L. Paolini. 2020. On Two Characterizations of Feature Models. In *ICTAC (LNCS)*, V. K. I. Pun, V. Stolz, and A. Simao (Eds.). Springer, 103–122.

[30] M. Davis, G. Logemann, and D. Loveland. 1962. A Machine Program for Theorem-Proving. *Comm. ACM* 5, 7 (1962), 394–397.

[31] M. Davis and H. Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (1960), 201–215.

[32] L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT solver. In *TACAS*. Springer, 337–340.

[33] DIMACS. 1993. Satisfiability: Suggested format.

[34] N. Eén and N. Sörensson. 2004. An Extensible SAT-solver. In *SAT*. Springer, 502–518.

[35] S. El-Sharkawy, A. Krafczyk, and K. Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *GPCE*. ACM, 45–54.

[36] A. Felfernig, D. Benavides, J. A. Galindo, and F. Reinfrank. 2013. Towards Anomaly Explanation in Feature Models. In *Proceedings of the International Configuration Workshop*. 117–124.

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

[38] J. K. Fichte, M. Hecher, and F. Hamiti. 2021. The Model Counting Competition 2020. *JEA* 26, Article 13 (2021), 26 pages.

[39] J. K. Fichte, M. Hecher, and V. Roland. 2022. Proofs for Propositional Model Counting. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 236)*, K. S. Meel and O. Strichman (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:24. https://doi.org/10.4230/LIPIcs.SAT.2022.30

[40] A. Flögel, H. Kleine Büning, and T. Lettmann. 1993. On the restricted equivalence for subclasses of propositional logic. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 27, 4 (1993), 327–340.

[41] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 91–100.

[42] E. Giunchiglia and R. Sebastiani. 2000. Applying the Davis-Putnam procedure to non-clausal formulas. In *AI*IA 99: Advances in Artificial Intelligence*, E. Lamma and P. Mello (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–94.

[43] J. Goubault-Larrecq and D. Plaisted. 2001. Normal form transformations. *Handbook of Automated Reasoning* 1 (2001), 273.

[44] M. Hentze, T. Pett, T. Thüm, and I. Schaefer. 2021. Hyper Explanations for Feature-Model Defect Analysis. In *VaMoS*. ACM, Article 14, 9 pages.

[45] R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* 27, 2 (2022), 1–34.

[46] T. Heß, C. Sundermann, and T. Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *SPLC*. ACM, 131–135.

[47] M. Iser, C. Sinz, and M. Taghdiri. 2013. Minimizing Models for Tseitin-Encoded SAT Instances. In *Theory and Applications of Satisfiability Testing – SAT 2013*, M. Järvisalo and A. Van Gelder (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 224–232.

[48] A. Israeli and D. G. Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *JSS* 83, 3 (2010), 485–501.

[49] P. Jackson and D. Sheridan. 2004. Clause Form Conversions for Boolean Circuits. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing* (Vancouver, BC, Canada) *(SAT'04)*. Springer-Verlag, Berlin, Heidelberg, 183–198.

[50] M. Janota. 2008. Do SAT Solvers Make Good Configurators?. In *SPLC*, Vol. 2. Lero Int. Science Centre, University of Limerick, 191–195.

[51] M. Järvisalo, A. Biere, and M. Heule. 2010. Blocked Clause Elimination. In *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–144.

[52] M. Järvisalo, A. Biere, and M. J. Heule. 2012. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning* 49, 4 (2012), 583–619.

[53] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (Mar. 2012), 89–92.

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

[55] C. Kästner. 2017. Differential Testing for Variational Analyses: Experience from Developing KConfigReader. arXiv:1706.09357 [cs.SE]

[56] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*. ACM, 805–824.

[57] C. H. P. Kim, D. Batory, and S. Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *AOSD*. ACM, 57—68.

[58] C. H. P. Kim, C. Kästner, and D. Batory. 2008. On the Modularity of Feature Interactions. In *GPCE*. ACM, 23–34.

[59] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *ESEC/FSE*. ACM, 291–302.

[60] M. Kowal, S. Ananieva, and T. Thüm. 2016. Explaining Anomalies in Feature Models. In *GPCE*. ACM, 132–143.

[61] S. Krieter, R. Arens, M. Nieke, C. Sundermann, T. Heß, T. Thüm, and C. Seidl. 2021. Incremental Construction of Modal Implication Graphs for Evolving Feature Models. In *SPLC*. ACM, 64–74.

[62] S. Krieter, R. Schröter, T. Thüm, W. Fenske, and G. Saake. 2016. Comparing Algorithms for Efficient Feature-Model Slicing. In *SPLC*. ACM, 60–64.

[63] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich. 2020. YASA: Yet Another Sampling Algorithm. In *VaMoS*. ACM, Article 4, 10 pages.

[64] S. Krieter, T. Thüm, S. Schulze, R. Schröter, and G. Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *ICSE*. ACM, 898–909.

[65] C. Kröher, S. El-Sharkawy, and K. Schmid. 2018. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 73–76.

[66] M. Krötzsch. 2010. *Description logic rules*. Vol. 8. IOS Press.

[67] A. Kübler, C. Zengler, and W. Küchlin. 2010. Model Counting in Product Configuration. In *LoCoCo*. Open Publishing Association, 44–53.

[68] E. Kuiter, A. Knüppel, T. Bordis, T. Runge, and I. Schaefer. 2022. Verification Strategies for Feature-Oriented Software Product Lines. In *VaMoS*. ACM, 12:1–12:9.

[69] E. Kuiter, S. Krieter, J. Krüger, K. Ludwig, T. Leich, and G. Saake. 2018. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 284–288.

[70] A. Kübler, C. Zengler, and W. Küchlin. 2010. Model Counting in Product Configuration. *Electronic Proceedings in Theoretical Computer Science* 29 (jul 2010), 44–53. https://doi.org/10.4204/eptcs.29.5

[71] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *SPLC*. Springer, 91–100.

[72] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. 2013. Scalable Analysis of Variable Software. In *ESEC/FSE*. ACM, 81–91.

[73] M. Lienhardt, F. Damiani, E. B. Johnsen, and J. Mauro. 2020. Lazy Product Discovery in Huge Configuration Spaces. In *ICSE*. ACM, 1509–1521.

[74] R. E. Lopez-Herrejon and D. Batory. 2001. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*. Springer, 10–24.

[75] P. Manolios and D. Vroon. 2007. Efficient Circuit to CNF Conversion. In *Theory and Applications of Satisfiability Testing – SAT 2007*, J. Marques-Silva and K. A. Sakallah (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 4–9.

[76] E. J. McCluskey. 1956. Minimization of Boolean functions. *The Bell System Technical Journal* 35, 6 (1956), 1417–1444.

[77] W. Meert, J. Vlasselaer, and G. Van den Broeck. 2016. A relaxed Tseitin transformation for weighted model counting. *Proceedings of the Sixth International Workshop on Statistical Relational AI (StarAI)*, 1–7.

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

[79] M. Mendonça, A. Wąsowski, and K. Czarnecki. 2009. SAT-Based Analysis of Feature Models is Easy. In *SPLC*. Software Engineering Institute, 231–240.

[80] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC*. ACM, 530–535.

[81] R. Muhammad and P. J. Stuckey. 2006. A stochastic non-CNF SAT solver. In *Pacific Rim International Conference on Artificial Intelligence*. Springer, 120–129.

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

[83] D.-J. Munoz, J. Oh, M. Pinto, L. Fuentes, and D. Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *SPLC*. ACM, 289–301.

[84] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *TSE* 41, 8 (2015), 820–841.

[85] J. A. Navarro-Pérez. N/A. Translations to propositional satisfiability. *Specification and Verification of Reconfiguration Protocols in Grid Component Systems* (N/A), 31.

[86] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *GPCE*. ACM, 188–201.

[87] A. Nonnengart, G. Rock, and C. Weidenbach. 1998. On generating small clause normal forms. In *Automated Deduction — CADE-15*, C. Kirchner and H. Kirchner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–411.

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

[89] J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers. 2020. *Scalable Uniform Sampling for Real-World Software Product Lines*. Technical Report TR-20-01. The University of Texas at Austin, Department of Computer Science.

[90] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. 2010. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *ICST*. IEEE, 459–468.

[91] T. Pett, S. Krieter, T. Runge, T. Thüm, M. Lochau, and I. Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *VaMoS*. ACM, Article 18, 9 pages.

[92] T. Pett, T. Thüm, T. Runge, S. Krieter, M. Lochau, and I. Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *SPLC*. ACM, 78–83.

[93] D. A. Plaisted and S. Greenbaum. 1986. A Structure-Preserving Clause Form Translation. *J. Symbolic Computation* 2, 3 (1986), 293–304.

[94] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST*. IEEE, 240–251.

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

[96] R. Pohl, K. Lauenroth, and K. Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *ASE*. IEEE, 313–322.

[97] H. Post and C. Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *ASE*. IEEE, 347–350.

[98] S. D. Prestwich. 2009. CNF Encodings. *Handbook of Satisfiability* 185 (2009), 75–97.

[99] W. V. Quine. 1952. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly* 59, 8 (1952), 521–531.

[100] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *RE*. IEEE, 136–145.

[101] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *ICSE*. ACM, 667–678.

[102] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. 2017. Compositional Analyses of Highly-Configurable Systems with Feature-Model Interfaces. In *SE*, J. Jürjens and K. Schneider (Eds.). Gesellschaft für Informatik, 129–130.

[103] S. Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *SPLC*, Vol. 2. IEEE, 201–207.

[104] A. A. Semenov. 2009. About Tseitin transformation in logical equations. *Prikladnaya Diskretnaya Matematika* 4 (2009), 28–50.

[105] S. Sharma, S. Roy, M. Soos, and K. S. Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter. In *IJCAI*, Vol. 19. AAAI Press, 1169–1176.

[106] S. She and T. Berger. 2010. *Formal Semantics of the Kconfig Language*. Technical Report. University of Waterloo.

[107] D. Sheridan. 2004. The Optimality of a Fast CNF Conversion and its Use with SAT. *SAT* 2 (2004).

[108] J. Sprey, C. Sundermann, S. Krieter, M. Nieke, J. Mauro, T. Thüm, and I. Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *VaMoS*. ACM, Article 6, 9 pages.

[109] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *VaMoS*. ACM, Article 12, 10 pages.

[110] C. Sundermann, T. Thüm, and I. Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *VaMoS*. ACM, Article 3, 9 pages.

[111] G. Sutcliffe and S. Melville. 1996. The practice of clausification in automatic theorem proving. (1996).

[112] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*. ACM, 47–60.

[113] G. Thomas. 2000. ECos: An Operating System for Embedded Systems. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25, 1 (2000), 66–72.

[114] T. Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *SPLC*. ACM, Article 16, 6 pages.

[115] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR* 47, 1 (2014), 6:1–6:45.

[116] T. Thüm, D. Batory, and C. Kästner. 2009. Reasoning About Edits to Feature Models. In *ICSE*. IEEE, 254–264.

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

[118] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. 2012. Family-Based Deductive Verification of Software Product Lines. In *GPCE*. ACM, 11–20.

[119] M. Thurley. 2006. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *SAT*. Springer, 424–429.

[120] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer, 466–483.

[121] C. Umans, T. Villa, and A. Sangiovanni-Vincentelli. 2006. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1230–1246.

[122] M. N. Velev. 2004. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference* (Yokohama, Japan) *(ASP-DAC '04)*. IEEE Press, 310–315.

[123] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. 2012. *Experimentation in Software Engineering*. Springer.

[124] W. Zhang, H. Zhao, and H. Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In *ICFEM*. Springer, 115–130.