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

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 \( \phi \) is a boolean variable \( x \), a negation \( \neg \phi \) of a formula \( \phi \) or a connection of two formulas \( \phi \) and \( \psi \) by one of the connectors and \( \phi \land \psi \), or \( \phi \lor \psi \), implies \( \phi \rightarrow \psi \), or biimplications \( \leftrightarrow \phi \leftrightarrow \psi \). A variable \( x \) and its negation \( \neg x \) are also referred to as literals. Implications \( \phi \rightarrow \psi \) and biimplications \( \leftrightarrow \phi \leftrightarrow \psi \) can be equivalently expressed as \( \neg \phi \lor \psi \) and \( (\neg \phi \lor \psi) \land (\phi \lor \neg \psi) \), respectively [23], which is why we focus on expressions using only the and or connectors. We refer to the set of all possible formulas as \( \Phi \). 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., \( \phi \) is of the form \( (x_{1,1} \lor \ldots \lor x_{1,m_1}) \land \ldots \land (x_{n,1} \lor \ldots \lor x_{n,m_n}) \), where all \( x_{i,j} \) are literals). Conjunctions and disjunctions can have an arbitrary number of arguments \( \geq 1 \), so \( x_1 \lor x_2 \) (one conjunct) and \( x_1 \land x_2 \) (two conjunctions with one disjunct each) are in CNF, too.

We evaluate the truth value (i.e., \( T \) 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 \( \langle \phi, \text{Var}_\phi \rangle \)), but assume that the set \( \text{Var}(\phi) \) contains all Boolean variables assignable in \( \phi \). An assignment for a formula \( \phi \) then consists of a subset of variables \( A \subseteq \text{Var}(\phi) \), each of which is implicitly set to \( T \). All other variables (i.e., those in \( \text{Var}(\phi) \setminus A \)) are implicitly set to \( \bot \). Given a formula \( \phi \) 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 \( \phi \) as \( \{\phi\} = \{A \mid A \subseteq \text{Var}(\phi), \phi(A) = T\} \).

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 \( \phi \) 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 \( \phi \) over the set of features \( F \) (i.e., \( F = \text{Var}(\phi) \)). A configuration is then represented by an assignment \( C \subseteq F \) for the feature-model formula \( \phi \). If an assignment \( C \) satisfies the feature dependencies given by \( \phi \) (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:

$$\phi_{\text{GFL}} = \text{GPL} \land \text{Edges} \land \text{Components} \land \text{Directed} \lor \text{Undirected} \land (\neg \text{Directed} \lor \neg \text{Undirected}) \land (\text{Cycles} \lor \text{Components}) \land (\neg \text{Components} \lor \text{Cycles}) \lor \text{Directed}$$

There exist three satisfying assignments for this formula:

\[
\{\text{GPL}, \text{Edges}, \text{Alg.}, \text{Dir.}, \text{Components}\}, \\
\{\text{GPL}, \text{Edges}, \text{Alg.}, \text{Dir.}, \text{Cycles}\}, \\
\{\text{GPL}, \text{Edges}, \text{Alg.}, \text{Dir.}, \text{Components}, \text{Cycles}\}
\]

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 \(\phi\), we define the void analysis as:

\[
\text{void}(\phi) \text{ if and only if } \{\phi\} = \emptyset
\]

If \(\phi\) is in CNF, we can implement this analysis by querying a SAT solver for \(\phi\). If \(\phi\) 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 \(\phi\) and a feature \(f\), we define:

\[
\text{dead}(\phi, f) \text{ if and only if } \{\phi \land f\} = \emptyset
\]

\[
\text{core}(\phi, f) \text{ if and only if } \{\phi \land \neg f\} = \emptyset
\]

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 \(\phi\) 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 \(\phi\) and a feature \(f\), we define feature-model cardinality \#fm(\(\phi\)) and feature cardinality \#f(\(\phi, f\)):

\[
\#\text{fm}(\phi) = |\{\phi\}|
\]

\[
\#f(\phi, f) = |\{\phi \land f\}|
\]

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 \#f(\(\phi, f\))/\#fm(\(\phi\)). In our example in Figure 1, the variability factor is \(\frac{3}{128} \approx 2.34\%\), 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 \(\frac{3}{2}\), 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 \(\phi\) that is in CNF, we can query a #SAT solver for \(\phi\). 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.
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 \( \theta : \Phi \rightarrow \Phi \) 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., \( \forall \phi \in \Phi : \text{Var}(\phi) \subseteq \text{Var}(\theta(\phi)) \)).

For feature-model analysis, we distinguish five properties that a CNF transformation can fulfill: equivalence, equi-satisfiability, equi-assignability, 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 \( \theta \) fulfills the property:

1. **Equivalence** [23] if the input formula \( \phi \) and the computed CNF \( \theta(\phi) \) are logically equivalent:
   \[
   \forall \phi \in \Phi : \text{Var}(\phi) = \text{Var}(\theta(\phi)) \land [\phi] = [\theta(\phi)]
   \]
   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 \( \theta \) preserves satisfiability between the input formula and the computed CNF:
   \[
   \forall \phi \in \Phi : [\phi] = \emptyset \Leftrightarrow [\theta(\phi)] = \emptyset
   \]
   Thus, either both formulas are satisfiable or none is.

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

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

5. **Quasi-equivalence** if \( \theta \) 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.

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

**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 \( \text{Var}(\phi) \) with \( \text{Var}(\theta(\phi)) \) 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 \( \phi \) is satisfiable precisely when \( [\phi] > 0 \).
- Equi-assignability implies quasi-equivalence, because both sets in the definition of equi-assignability are empty precisely when \( [\phi] \) and \( [\theta(\phi)] \) are.

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 \( \theta \) a CNF transformation that fulfills quasi-equivalence. Then, \( \phi \) and \( \theta(\phi) \) have isomorphic satisfying assignments (i.e., there is a bijection between them).

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

Theorem 3.4 justifies practical usage of quasi-equivalence, as \( \sigma \) allows us to convert between satisfying assignments of the original formula \( \phi \) 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, \( \theta \) 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 \( \phi \), 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, \( \theta \) a CNF transformation, and \( f \in \text{Var}(\phi) \) a feature. If \( \theta \) fulfills:

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

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 [106], 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 \( \lor \) can be interchanged with conjunctions \( \land \) (by distributivity) to create a conjunction of disjunctions of literals and thus a CNF.

#### 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 sub-formula \( \phi \) of a non-CN F-formula with a new Boolean variable \( x_\phi \), which acts as a “shortcut” to refer to \( \phi \). The new variable \( x_\phi \) is then tied to the original subformula \( \phi \) using a bijimpression \( \leftrightarrow \). 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.

---

1Claim (1) and (3) hold by definition of equi-satisfiability and equi-countability. Claim (2) is proven by Schröter et al. [106, Theorem 15]. Claim (4) follows from the same theorem by using the bijection \( \sigma \) for brevity; we omit a proof.
We propose a succinct definition along the lines of Bradley and Manna [21] to illustrate the procedure for binary ∧ and ∨.

Definition 4.2. We write \( \psi \subseteq \varphi \) to denote that \( \psi \) is a subformula of \( \varphi \). The Tseitin transformation \( \theta_T \) is then given by the NNF step \( \text{nnf} \) followed by the replacement step \( \text{rep}_\theta \) given by

\[
\text{rep}_\theta(\varphi) := \langle \varphi \rangle \land \bigwedge_{\varphi \in \varphi}[\varphi \land \varphi] \text{classify}(\varphi \land \varphi) \leftrightarrow ((\varphi) + ((\varphi) + (\varphi))
\]

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

and classify looks up a precomputed CNF expression. \(^2\) So, \( \theta_T := \text{rep}_\theta \circ \text{nnf} \) (i.e., transform into NNF first, then into CNF by introducing auxiliary variables \( x_\theta \) that are tied to subformulas \( \varphi \)).

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

\[
\theta_T(\phi_{\text{GPL}}) = x_1 \\
\land \text{classify}(x_1 \leftrightarrow (G \land E \land A \land x_2 \land x_3 \land x_4 \land x_5)) \\
\land \text{classify}(x_2 \leftrightarrow (D \land U)) \land \text{classify}(x_3 \leftrightarrow (-D \lor -U)) \\
\land \text{classify}(x_4 \leftrightarrow (Co \lor Cy)) \land \text{classify}(x_5 \leftrightarrow (x_6 \lor D)) \\
\land \text{classify}(x_6 \leftrightarrow (-Co \lor -Cy))
\]

In \( \theta_T(\phi_{\text{GPL}}) \), we represent each non-trivial subformula \( \psi \subseteq \phi_{\text{GPL}} \) by assigning it an auxiliary variable \( \langle \psi \rangle = x_1 \) 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 classify, but in practice, they can be directly generated in CNF. Finally, to "activate" the entire formula \( \theta_T(\phi_{\text{GPL}}) \), we add its root auxiliary variable \( \langle \phi_{\text{GPL}} \rangle = x_1 \) as a unit clause.

As this example illustrates, the Tseitin transformation \( \theta_T \) introduces new variables and may therefore result in formulas larger than necessary. However, contrary to \( \theta_D \), the Tseitin transformation \( \theta_T \) 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 \( \theta_T \) does not fulfill equivalence (cf. Section 3.1), as it introduces new variables. However, it has been proven that \( \theta_T \) fulfills both equi-assignability \([93]\) and equi-countability \([104]\). So, \( \theta_T \) 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). 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 \( \theta_{PG} := \text{rep}_\theta \circ \text{nnf} \) (i.e., transform into NNF first, then into CNF by introducing variables \( x_\theta \) that imply some subformula \( \phi \)).

It has been proven \([93]\) that the Plaisted-Greenbaum transformation \( \theta_{PG} \) introduces fewer clauses than the Tseitin transformation \( \theta_T \) 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, \( \theta_{PG} \) also has a disadvantage compared to \( \theta_T \): By using \( \rightarrow \) instead of \( \leftrightarrow \), \( \theta_{PG} \) only encodes the satisfiability of subformulas, so the number of satisfying assignments is not preserved by the transformation. Thus, \( \theta_{PG} \) 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 (a) it 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:

**RQ1:** How efficient are CNF transformation tools?

**RQ2:** Does the efficiency of subsequent feature-model analyses depend on the CNF transformation tool?

**RQ3:** 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...
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\(^2\) 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\(^5\) [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\(^6\) [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.

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

<table>
<thead>
<tr>
<th>Source</th>
<th>System</th>
<th>Version</th>
<th>#Var</th>
<th>#Lit</th>
</tr>
</thead>
<tbody>
<tr>
<td>KConfig</td>
<td>Fiasco</td>
<td>58a50a</td>
<td>85/116</td>
<td>554/856</td>
</tr>
<tr>
<td>(extracted)</td>
<td>Kinec</td>
<td>6.4</td>
<td>123/185</td>
<td>640/1227</td>
</tr>
<tr>
<td></td>
<td>uClibc-ng</td>
<td>1.0.40</td>
<td>246/388</td>
<td>3874/8421</td>
</tr>
<tr>
<td></td>
<td>BusyBox</td>
<td>1.35.0</td>
<td>866/986</td>
<td>5159/4728</td>
</tr>
<tr>
<td></td>
<td>EmbToolkit</td>
<td>1.8.0</td>
<td>2510/4321</td>
<td>28k/108k</td>
</tr>
<tr>
<td></td>
<td>Buildroot</td>
<td>2021.11.2</td>
<td>7548/9228</td>
<td>215k/584k</td>
</tr>
<tr>
<td></td>
<td>Freetz-NGL</td>
<td>5c54d1</td>
<td>9k/11k</td>
<td>1119k/2713k</td>
</tr>
<tr>
<td></td>
<td>Linux (x86)</td>
<td>4.18</td>
<td>13k/22k</td>
<td>223k/1072k</td>
</tr>
<tr>
<td>KConfig</td>
<td>axTLS</td>
<td>N/A</td>
<td>96</td>
<td>414</td>
</tr>
<tr>
<td>(hierarchy)</td>
<td>uClibc</td>
<td>N/A</td>
<td>313</td>
<td>2739</td>
</tr>
<tr>
<td></td>
<td>uClinux-base</td>
<td>N/A</td>
<td>380</td>
<td>15k</td>
</tr>
<tr>
<td></td>
<td>BusyBox</td>
<td>1.18.0</td>
<td>854</td>
<td>2666</td>
</tr>
<tr>
<td></td>
<td>EmbToolkit</td>
<td>N/A</td>
<td>1179</td>
<td>35k</td>
</tr>
<tr>
<td></td>
<td>uClinux-dist</td>
<td>N/A</td>
<td>1580</td>
<td>4201</td>
</tr>
<tr>
<td></td>
<td>Linux (x86)</td>
<td>2.6.33.3</td>
<td>6467</td>
<td>45k</td>
</tr>
<tr>
<td>CDL (3/116)</td>
<td>Min: am31_sim</td>
<td>N/A</td>
<td>1178</td>
<td>5669</td>
</tr>
<tr>
<td></td>
<td>Median: lpcm</td>
<td>N/A</td>
<td>1262</td>
<td>6167</td>
</tr>
<tr>
<td></td>
<td>Max: ea2468</td>
<td>N/A</td>
<td>1408</td>
<td>6908</td>
</tr>
<tr>
<td>Closed-Source</td>
<td>Automotive</td>
<td>2.1-2.4</td>
<td>14k-19k</td>
<td>485k-714k</td>
</tr>
</tbody>
</table>

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.\(^4\)

1Automatic scripts available at: https://doi.org/10.5281/zenodo.6922807
2Replication package available at: https://doi.org/10.5281/zenodo.6525375
3https://github.com/torvalds/linux/blob/master/Documentation/kbuild/kconfig-
language.rst
4https://github.com/ckaestne/kconfigreader
5https://github.com/paulgazz/kmax
6https://github.com/_StyleChef
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 $\theta_D$ with some optimizations (e.g., subsumption of clauses and optional unit propagation). 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 $\theta_T$. That is, tseitin-cnf introduces an auxiliary variable for a subformula $\phi \subseteq \psi$ if $\psi$ occurs several times in $\phi$ or the exponential blowup by the distributive transformation would be unacceptable—otherwise, it uses the distributive transformation on $\psi$. 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 bimplications $\leftrightarrow$ to replace subformulas [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 $\theta_P$. Similar to Z3, KConfigReader only introduces an auxiliary variable for a subformula $\phi \lor \psi$ if the predicted number of clauses for the distributive transformation $|\phi| + |\psi|$ exceeds a fixed threshold. 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 $\phi$, we construct DIMACS files, containing the following:

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

Table 2: Evaluated SAT and #SAT solvers.

<table>
<thead>
<tr>
<th>Class Year</th>
<th>Solver</th>
<th>Class Year</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>2002∗, 04†</td>
<td>SAT</td>
<td>2013−14∗</td>
</tr>
<tr>
<td>2003∗</td>
<td>Forklift</td>
<td>2016−19∗</td>
<td>MapleSAT</td>
</tr>
<tr>
<td>2005†</td>
<td>SatELateGTT</td>
<td>2020−21†</td>
<td>Kinsat</td>
</tr>
<tr>
<td>2006†</td>
<td>MiniSat</td>
<td>2020†</td>
<td>countAntom</td>
</tr>
<tr>
<td>2007†</td>
<td>RSat</td>
<td>2021†</td>
<td>d4</td>
</tr>
<tr>
<td>2009†</td>
<td>PrecosAT</td>
<td>-</td>
<td>dSharp</td>
</tr>
<tr>
<td>2010†</td>
<td>CryptoMiniSat</td>
<td>2020†</td>
<td>Ganak</td>
</tr>
<tr>
<td>2011−12‡</td>
<td>Glucose</td>
<td>2021‡</td>
<td>sharpSAT</td>
</tr>
</tbody>
</table>

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

(3) $\phi \land \lnot f$ 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 (RQ1) and the performed feature-model analyses (RQ2). 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.

11http://www.satcompetition.org/
12https://mccompetition.org/
which causes an exponential growth when transforming specific KConfigReader

turesIDE are likely due to its use of the distributive transformation, when

succeeds, it is significantly (1359 ms, and 223 ms, respectively. Overall, in the cases where

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

which 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 RQ2: 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 95% of 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>
<thead>
<tr>
<th>RQ</th>
<th>Compared CNF Transformations</th>
<th>p-Value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1 FeatureIDE KConfigReader</td>
<td>8.73 × 10^{-6}</td>
<td>-3.12</td>
<td></td>
</tr>
<tr>
<td>FeatureIDE Z3</td>
<td>2.90 × 10^{-3}</td>
<td>-9.34 × 10^{-1}</td>
<td></td>
</tr>
<tr>
<td>KConfigReader Z3</td>
<td>1.69 × 10^{-2}</td>
<td>6.20 × 10^{-1}</td>
<td></td>
</tr>
<tr>
<td>RQ2 FeatureIDE KConfigReader</td>
<td>4.20 × 10^{-2}</td>
<td>-5.44 × 10^{-2}</td>
<td></td>
</tr>
<tr>
<td>FeatureIDE Z3</td>
<td>4.20 × 10^{-2}</td>
<td>-5.44 × 10^{-2}</td>
<td></td>
</tr>
<tr>
<td>KConfigReader Z3</td>
<td>8.60 × 10^{-1}</td>
<td>-3.91 × 10^{-3}</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Results of paired t-test on absolute runtime.
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 RQ3: 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.

<table>
<thead>
<tr>
<th>Analysis/CNF Trans.</th>
<th>FeatureIDE</th>
<th>Z3</th>
<th>KConfigReader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Dead/Core</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>FMC</td>
<td>100%</td>
<td>100%</td>
<td>29.412%</td>
</tr>
<tr>
<td>FC</td>
<td>100%</td>
<td>100%</td>
<td>29.502%</td>
</tr>
</tbody>
</table>

By switching the CNF transformation, this showcases that the transformation tool is relevant for the runtime of SAT- and #SAT-based analyses using the transformed CNFs as input. We note that 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.

**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 (RQ3).](image-url)
While it is well-recognized in automated reasoning that CNF en-
we perform a practical evaluation of concrete tools for CNF trans-
which we manually confirmed for our evaluation.

(e.g., by comparing cardinalities). In particular, the #SAT solver
variables) and sources (extracted models and hierarchies). Second,
Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses ASE ’22, October 10–14, 2022, Rochester, MI, USA
Järvisalo et al. [51] compare the Tseitin and Plaisted-Greenbaum
transformations can be regarded as a kind of Tseitin transformation.
adds abstract features to eliminate complex constraints; this trans-
slicing [29, 73, 102]. Slices are excerpts of feature models that do not
properties on other feature-model transformations, such as slicing.
end, we want to investigate further what makes a constraint difficult
to transform. We also aim to apply our taxonomy of 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 practi-
tioners 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 transforma-
tion itself, but also on the efficiency and even the correctness of
subsequent analyses. In particular, there seems to be a tradeoff be-	ween 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 KCONFi-
gREADER 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 eval-
uation 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.
Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses

ASE ’22, October 10–14, 2022, Rochester, MI, USA


[63x212]GPCE

Texas at Austin, Department of Computer Science.

[64x228](N/A), 31.


[72x316]ICST

Scalable Analysis of Variable Software. In

[73x324]VaMoS

Applications of #SAT Solvers on Feature Models. In

[74x336]VaMoS

Strategies for Feature-Oriented Software Product Lines. In

[75x356]VaMoS

Planning and Survey of Analysis Strategies for Software Product Lines. In

[76x372]VaMoS

and Survey of Analysis Strategies for Software Product Lines. In

[77x395]VaMoS

and Survey of Analysis Strategies for Software Product Lines. In

[78x403]VaMoS

and Survey of Analysis Strategies for Software Product Lines. In

[79x419]The Bell System Technical Journal


[81x451]. Springer, 10–24.

[82x451]. Springer, 10–24.

[83x473], C. Kirchner and H. Kirchner

[84x507], Vol. 2. IEEE, 201–207.


[86x547]. Springer, 10–24.

[87x555]. IEEE, 136–145.


[89x587]. Springer, 91–100.

[90x603]. Springer, 424–429.

[91x611]. ACM, 667–678.

[92x651]. IEEE, 136–145.


[95x682]. Springer, 91–100.

[96x690]. ACM, 1509–1521.

[97x700]. ACM, 1509–1521.


[99x726]. Springer, 231–240.

[100x742]. Springer, 4–9.


[105x795]. Springer, 201–207.

[106x807]. ACM, 188–201.

[107x813]. ACM, 188–201.


[109x826]. Springer, 91–100.

[110x832]. Springer, 231–240.


[112x844]. Springer, 231–240.

[113x850]. Springer, 231–240.

[114x856]. Springer, 231–240.

[115x862]. Springer, 231–240.


[118x880]. Springer, 231–240.


[120x892]. Springer, 231–240.

[121x898]. Springer, 231–240.

[122x904]. Springer, 231–240.


[126x928]. Springer, 231–240.

[127x934]. Springer, 231–240.


[129x946]. Springer, 231–240.

[130x952]. Springer, 231–240.

[131x958]. Springer, 231–240.

[132x964]. Springer, 231–240.

[133x970]. Springer, 231–240.

[134x976]. Springer, 231–240.