Is There a Mismatch between Real-World Feature Models and Product-Line Research?

Alexander Knüppel
TU Braunschweig, Germany
a.knueppel@tu-bs.de

Thomas Thüm
TU Braunschweig, Germany
t.thuem@tu-bs.de

Stephan Mennicke
TU Braunschweig, Germany
mennicke@ips.cs.tu-bs.de

Jens Meinicke
University of Magdeburg, Germany
Carnegie Mellon University, USA
meinicke@ovgu.de

Ina Schaefer
TU Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

Feature modeling has emerged as the de-facto standard to compactly capture the variability of a software product line. Multiple feature modeling languages have been proposed that evolved over the last decades to manage industrial-size product lines. However, less expressive languages, solely permitting require and exclude constraints, are permanently and carelessly used in product-line research. We address the problem whether those less expressive languages are sufficient for industrial product lines. We developed an algorithm to eliminate complex cross-tree constraints in a feature model, enabling the combination of tools and algorithms working with different feature model dialects in a plug-and-play manner. However, the scope of our algorithm is limited. Our evaluation on large feature models, including the Linux kernel, gives evidence that require and exclude constraints are not sufficient to express real-world feature models. Hence, we promote that research on feature models needs to consider arbitrary propositional formulas as cross-tree constraints prospectively.

CCS CONCEPTS

• **Software and its engineering** → **Feature interaction**; **Software product lines**;

KEYWORDS

Software product lines, feature modeling, cross-tree constraints, model transformation, expressiveness, require constraints, exclude constraints

ACM Reference Format:
Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch between Real-World Feature Models and Product-Line Research?. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 04-08, 2017,* 12 pages.
https://doi.org/10.1145/3106237.3106252

1 INTRODUCTION

Software product-line engineering is a paradigm enabling mass customization of software [30]. Instead of developing a monolithic software product, the goal is to develop reusable software artifacts for a specific domain in a process called *domain engineering*. Multiple software artifacts composed together eventually result in a software product. A *software product line* is a family of similar software products sharing common artifacts. We distinguish between common and varying characteristics of products in terms of *features*. Features are user-visible aspects or characteristics of a software [22], being of interest for some stakeholders. Later, in a process called *application engineering*, a set of features is selected based on the requirements of stakeholders and a software product is derived.

The standard technique in research and industry to manage variability of a product line is *feature modeling* [12, 22]. Feature models offer an easy-to-understand formalism and unambiguously describe dependencies among features. In the context of product-line engineering, feature modeling is a valuable asset in several areas such as domain scoping [12, 22], feature-oriented software development [22, 24, 42], product-line analysis [39], and configuration management [48]. Our ten year experience with developing the open-source tool FeatureIDE [24] and integrating product-line tools is that a typical obstacle is the expressive power of different feature modeling dialects. Varying expressiveness in feature modeling languages prevents tool reuse and, thus, hinders efficient application of existing algorithms and concepts.

Over the last decades, several feature modeling languages, extending the initially proposed language by Kang et al. [22], have been suggested, either graphical [6, 12, 16, 18, 20, 23, 24, 31] or textual [2, 4, 5, 8, 10, 24, 28, 32, 44]. Ideally, given a set of features, a feature modeling language should be able to represent exactly the set of all valid feature combinations with respect to the requirements aquired during the domain engineering phase. A considerable portion of such languages, however, is not *expressively complete* (i.e., in theory, certain product lines cannot be represented). Although the restricted expressiveness was mentioned elsewhere [14, 17, 33, 37], an in-depth analysis of the problem for real-world feature models and a practical solution to overcome this limitation are still missing.

In particular, we identified several proposed methods dealing with feature models that are still based on expressively incomplete languages due to their simplicity and dominance in the product-line community. To name a few, the affected reasearch areas include *automated analysis of feature models* [34], *synthesis of feature* *models* [3, 21, 26, 27, 37], *product-line testing and analysis* [15, 38], *generation of artificial feature models for experiments and evaluations* [19, 35], and *optimal feature selection* [6, 19, 47, 48]. More surprisingly, the number of anually proposed methods that are based on expressively incomplete feature modeling languages does not seem to decrease over time, as we still identified several publications in the years 2015 – 2017 (e.g., [11, 27, 36, 45, 46]).

Typically, expressively incomplete languages used in product-line research facilitate only two kinds of cross-tree constraints, here called *simple constraints*: either the activation of a feature f_1 implies the activation of a feature f_2 (i.e., f_1 *requires* f_2) or the two features are mutually exclusive and cannot be activated together (i.e., f_1 *excludes* f_2) [22]. We refer to feature models facilitating only simple constraints as *basic feature models*.

In contrast to simple constraints, *complex constraints* are arbitrary propositional formulas over the set of features written as textual constraints [5]. Complex constraints are already part of many feature modeling languages used in practice, such as FeatureIDE[24], Familiar[2], or Clafer[4]. Other variability languages, such as KConfig and CDL, where feature model approximations exist [9], also rely on flavors of propositional logic to document dependencies between features across the feature model hierarchy.

To overcome the problem of different languages required at different stages in the engineering process, a feature model transformation is necessary. However, feature models with complex constraints cannot generally be transformed into ones with only simple constraints, as their languages differ in expressive power. Nevertheless, to answer the question whether there is a mismatch between real-world feature models and product-line research, we need to bridge the gap between those different languages.

We propose *relaxed feature models*, an expressively complete language based on simple constraints. In theory, this language can replace basic feature models for various methods in product-line research. However, relaxed feature models may increase significantly in the number of features and constraints. We analyze the usefulness of this transformation on real-word feature models. In particular, the contributions of this paper are as follows.

- We provide examples of product-line research solely focusing on basic feature models.
- We present a product-preserving transformation from languages using complex constraints to relaxed feature models, and formally prove its correctness.
- We quantitatively assess the limited expressiveness of feature models with only simple constraints.
- We give evidence that real-world feature models rely on complex constraints.
- We evaluate our transformation on large real-world feature models and discuss consequences for product-line research.

2 EXPRESSIVENESS OF FEATURE MODELS IN PRODUCT-LINE RESEARCH

In this section, we introduce *basic feature models*, a language predominantly used in product-line research. Thereupon, we investigate its expressive power. A basic feature model is a hierarchically organized tree structure that decomposes features into either an or-group, an alternative-group, or sole mandatory and optional features. Furthermore, depending on the language, require and exclude constraints can be specified [12, 22]. In Figure 1, we exemplify a basic feature model representing a product line for *pizzas*.

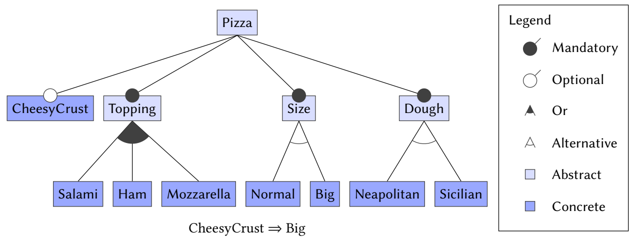

Figure 1: Basic feature model representing a product line for pizzas in FeatureIDE notation.

Features *Topping*, *Size*, and *Dough* are mandatory, thus are part of all pizza products. As for toppings, we must at least select one of the features *Salami, Ham*, or *Mozzarella*. Regarding the size, we may either choose feature *Normal* or *Big*. We can also decide which dough we would like to use, namely classic *Neapolitan* or *Sicilian*. Finally, we can decide to get extra cheese inside our crust, offered by the optional feature *CheesyCrust*. Nevertheless, by fulfilling the requires constraint depicted below the diagram (i.e., *CheesyCrust* ⇒ *Big*), we force the size of a pizza to be big, whenever we order a cheesy crust.

A feature model language can be described informally by a concrete syntax (i.e., as we did above), or formally by defining a formal model. We are interested in describing our approach formally to precisely and umambigiously reason about our aforementioned contributions. For this purpose, Schobbens et al. [33] proposed a generic formal semantics to catch a variety of older feature modeling dialects. To increase expressiveness, they used *directed acyclic graphs* instead of trees. However, in our experience, the vast majority of feature modeling languages build upon a tree structure. Thus, we give a modified version of the semantics for basic feature models compared to the one Schobbens et al. proposed [33]. More precisely, we obtained the requirements for our basic feature modeling language by conducting an expert survey of scientific publications in product-line research. Table 1 lists 15 publications categorized by their respective product-line discipline that go beyond the analysis of propositional logic, for which it is not obvious how they can be used for feature models with complex constraints.

We identified three common characteristics of a basic feature modeling language: (1) features are only decomposed into optional features, mandatory features, or-groups, and alternative-groups, (2) the hierarchy is built upon a tree instead of a directed acyclic graph, and (3) only require and exclude constraints (i.e., simple constraints) are allowed. Optional and mandatory features below one parent are typically grouped together into an *and-group*.

Table 1: Summary of reviewed publications using basic feature models for five application domains.

Research Area	Proposal
Analysis of feature models	[11, 34, 46]
Feature model synthesis	[3, 21, 26, 27, 37]
Generating artificial feature models	[19, 35]
Product-line testing	[15, 38]
Optimal feature selection	[6, 19, 47, 48]

We denote by $\mathcal{F}$ the universe of features. Given a set of features $N \subseteq \mathcal{F}$, we distinguish between *concrete features* that are mapped to software artifacts (i.e., $P \subseteq N$) and *abstract features* [41] (i.e., $N \setminus P$) that are either used for grouping and decomposition or that are planned to be connected with software artifacts later during software evolution. Abstract features have the advantage that each decomposition belongs to exactly one feature and groups can be explicitly labeled (cf. Figure 1 where features *Pizza*, *Topping*, *Size*, and *Dough* are abstract for grouping their sub-features). In other concrete syntaxes without abstract features, it is possible to decompose a feature into multiple groups (e.g., or- and alternative-groups) without intermediate features [12]. Both approaches can be used interchangebly with respect to the set of valid products [41]. Inspired by the formal semantics of Schobbens et al. [33], we define the syntactic domain of basic feature models as follows.

Definition 2.1. A basic feature model is defined as a 7-tuple $(N, P, r, \omega, \lambda, \Pi, \Psi)$ where

- $N \subseteq \mathcal{F}$ is a finite set of features and $P \subseteq N$ a subset of concrete features.
- $r \in N$ is the root feature.
- $\omega : N \to \{0, 1\}$ is a function labeling a feature as either optional (0) or mandatory (1) with $\omega(r) = 1$.
- $\lambda : N \to \mathbb{N} \times \mathbb{N}$ is a function representing the relationship of a parent feature and its sub-features. The lower bound is the minimal number of features that must be selected, and the upper bound is the maximal number of features that can be selected. We use $\langle 1..1 \rangle$ for alternative-groups, $\langle 1..n \rangle$ for or-groups with n sub-features, $\langle n..n \rangle$ for and-groups with n sub-features, and $\langle 0..0 \rangle$ for leaf features.
- $\Pi \subseteq N \times N$ is a decomposition relation. We denote $(f, g) \in \Pi$ as $f \prec g$, meaning that g is sub-feature of f.
- $\Psi \subseteq \{f \Rightarrow g, f \Rightarrow \neg g \mid f, g \in P\}$ is a set of simple constraints in propositional logic.

A basic feature model has an acyclic tree structure (i.e., except for root feature r, every feature has exactly one parent) and leaf features must be concrete (i.e., $\forall f \in N$, if $\lambda(f) = \langle 0..0 \rangle$ then $f \in P$). Moreover, only features of an and-group can be mandatory (i.e., for $f, g \in N$, if $g \prec f$ and $\omega(f) = 1$, then $\lambda(g) = \langle n..n \rangle$ with n being the number of sub-features of g). The set of all basic feature models is denoted by $\mathcal{L_B}$.

Example 2.2. Consider a feature model $(N, P, r, \omega, \lambda, \Pi, \Psi) \in \mathcal{L_B}$ as depicted in Figure 2. The representation in $\mathcal{L_B}$ is illustrated in the following, where names of features are abbreviated by their first identifying letters.

$$
\begin{array}{lcl}
P & = & \{Sa, H, M, N, B\} \\
N & = & \{Pi, T, Si\} \cup P \\
r & = & Pi \\
\omega(f) & = & \begin{cases} 0 & \text{if } f \in \{Sa, H, M, N, B\} \\ 1 & \text{if } f \in \{Pi, T, Si\} \end{cases} \\
\lambda(f) & = & \begin{cases} \langle 0..0 \rangle & \text{if } f \in \{Sa, H, M, N, B\} \\ \langle 2..2 \rangle & \text{if } f = Pi \\ \langle 1..1 \rangle & \text{if } f \in \{Si\} \\ \langle 1..3 \rangle & \text{if } f = T \end{cases} \\
\Pi & = & \{(Pi, T), (Pi, Si), (T, Sa), (T, H), \\
 & & (T, M), (Si, N), (Si, B)\} \\
\Psi & = & \{B \Rightarrow M\}
\end{array}
$$

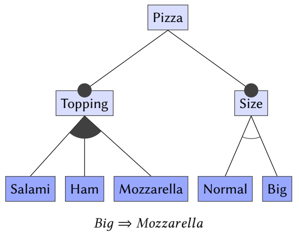

Figure 2: Reduced feature model for the pizza product line.

The distinction between concrete and abstract features allows us to define the semantics of basic feature models only considering features that influence the final product. For instance, in Example 2.2, features *Pizza*, *Topping*, *Size*, and *Dough* are abstract such that their integration into any valid program variant has no direct effect. Nevertheless, they are used in basic feature models for grouping and enabling the selection of sub-features. This is particularly important when comparing two or more feature models, as they may syntactically differ, but still represent the same software product line [40, 41].

Therefore, it is important to distinguish features in *configurations* (i.e., a feature selection possibly including abstract features) and features in *program variants* (i.e., the list of concrete features as an abstraction from implementation details). To retrieve an abstraction of a program variant from a given configuration, we must remove all abstract features. Therefore, we define the set of *valid configurations* as follows.

Definition 2.3. Let $m = (N, P, r, \omega, \lambda, \Pi, \Psi)$ be a basic feature model. Configuration $c \in 2^N$ is *valid* for m, denoted by $c \models_C m$, if and only if

- it contains the root feature: $r \in c$.
- it satisfies the decomposition type: $\forall f \in c$, $\lambda(f) = \{\langle 0..0 \rangle \ \langle 1..1 \rangle \ \langle 1..n \rangle \ \langle n..n \rangle\}$, where n is the number of sub-features of f, and $mand(f) \subseteq c$ must hold, where $mand(f) = \{ g \in N \mid \omega(g) = 1 \wedge f \prec g \}$ is the set of mandatory sub-features of f.
- its parent-child-relationships hold:
$$\forall f \in c : f' \prec f \text{ implies } f' \in c, \text{ and}$$
- it satisfies each cross-tree constraint:
$$\forall \psi \in \Psi : \bigwedge_{f \in c} f \wedge \bigwedge_{f' \in N \setminus c} \neg f' \models \psi$$

We denote the set of all valid configurations of m by C_m.

Based on Definition 2.3, the *semantic function* maps a feature model in $\mathcal{L_B}$ to its product line. The semantic domain $\mathcal{D}$ (i.e., the set of all existing product lines) is defined as $\mathcal{D} = 2^{2^{\mathcal{P}}}$ with $\mathcal{P} \subseteq \mathcal{F}$ being the set of concrete features. For basic feature models, we denote by $\mathcal{D_B} \subseteq \mathcal{D}$ the semantic domain of $\mathcal{L_B}$.

Definition 2.4. The semantics of a basic feature model m is its set of *valid program variants*, defined by $[\![m]\!]_{\mathcal{B}} := \{c \cap P \mid c \in C_m\}$.

Example 2.5. Consider the feature model $m \in \mathcal{L}_B$ in Figure 2 inspired by the pizza product line. The semantic function based on

Definition 2.4 results in the following product line, comprising 11 program variants in total.

$$
\begin{aligned}
[\![m]\!]_{\mathcal{B}} = \{&\{N, Sa\}, \{N, H\}, \{N, M\}, \{N, Sa, H\},\\
&\{N, Sa, M\}, \{N, H, M\}, \{N, Sa, H, M\},\\
&\{B, M\}, \{B, Sa, M\}, \{B, H, M\}, \{B, Sa, H, M\}\}
\end{aligned}
$$

Conversely to Example 2.5, an interesting problem is whether a representing feature model for a given product line exists. For this purpose, we define a language as *expressively complete*, if the domain of the language equals $\mathcal{D}$ and *expressively incomplete* otherwise. Whether a basic feature modeling language is sufficient to express all theoretical product lines is answered by the following theorem. For convenience, the product line of interest is visualized in Figure 3 by the left feature model with two complex constraints.

Theorem 2.6. *The language of basic feature models $\mathcal{L}_B$ is expressively incomplete (i.e., $\mathcal{D}_{\mathcal{B}} \neq \mathcal{D}$).*

Proof. It is sufficient to only show one product line π for which no basic feature model exists. We choose $\pi = \{\{A, B\}, \{A, C\}, \{B\}, \{B, C\}, \{A, B, C\}\}$. Based on Definition 2.1, features of m can have optional and mandatory features, or-groups, or alternative-groups below them. Furthermore, simple constraints can be specified. We make the following observations.

- *Parent-child-relationships and constraints:* No feature is occurring with any other feature in every product. Hence, there are neither parent-child-relationships nor require constraints between features A, B, and C. Product $\{A, B, C\}$ further reveals that there are no exclude constraints between these three features.
- *Alternative-groups:* Similar to above, product $\{A, B, C\}$ reveals that there are no alternative-groups.
- *Mandatory features:* There is no single feature occurring in every product. Therefore, neither feature A, B, nor C are mandatory sub-features of r.
- *Optional features:* Features A, B, and C cannot all be optional sub-features of r, because the empty product is missing.
- *Or-groups and abstract features:* Assume $f_1, f_2 \in \{A, B, C\}$ with $f_1 \neq f_2$ are in the same or-group. Since there are no cross-tree constraints, no parent-child-relationships, no alternative-groups, and no mandatory features, $\{\{f_1\}, \{f_2\}\} \subset \pi$ must hold. This, however, is contradicting to the product line π. Abstract features do not improve the situation, since they can only be placed above the or-group or be part of the or-group with the remaining third feature below. No options left are enough to represent the product line.

□

The other two feature models in Figure 3 are further examples of feature models where no pendant in $\mathcal{L}_B$ exist. Theorem 2.6 proves that, in theory, methods and tools in product-line research limit their applicability if they only consider basic feature models. However, it is unclear whether real-world feature models are affected by this limitation. Hence, we formally investigate expressively complete languages used for real-world feature models in the next section.

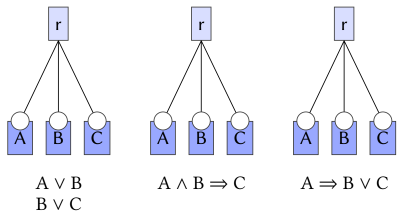

Figure 3: Three small feature models using complex constraints that cannot be expressed with the basic feature modeling language $\mathcal{L}_{\mathcal{B}}$.

3 EXPRESSING REAL-WORLD FEATURE MODELS

This section introduces (1) feature models with complex constraints, and (2) *relaxed feature models*, whereas the latter may serve as a substitution for basic feature models in product-line research.

3.1 Feature Models with Complex Constraints

Complex constraints are arbitrary propositional formulas over the set of features. In Definition 2.1, we already encoded simple constraints in $\mathcal{L}_{\mathcal{B}}$ using propositional logic. Hence, complex constraints can be seen as a generalization, since we now allow any logical connection between an arbitrary number of features. Consequently, semantic function $[\![.]\!]_{\mathcal{B}}$ (cf. Definition 2.4) carries over to both languages. For convenience, we simply use $[\![.]\!]$ in the following. We define the syntactic domain $\mathcal{L}_{\mathcal{M}}$ based on complex constraints as follows.

Definition 3.1. A *feature model* in $\mathcal{L}_M$ is a 7-tuple $(N, P, r, \omega, \lambda, \Pi, \Psi)$ where

- $N, P, r, \omega, \lambda, \Pi$ follow Definition 2.1 and
- cross-tree constraints are arbitrary propositional formulas over the set of features N, i.e., $\Psi \subseteq \mathbb{B}(N)$.

Theorem 3.2. *The language $\mathcal{L}_{\mathcal{M}}$ is expressively complete.*

Proof. Let $\pi \in \mathcal{D}$ be a product line. We construct feature model $m = (P \cup \{r\}, P, r, \omega, \lambda, \Pi, \Psi)$ such that $m \in \mathcal{L}_{\mathcal{M}}$ with $P = \bigcup_{p \in \pi} p$ and each feature $f \in P$ holds the following conditions.

- is sub-feature of root r: $r \prec f$.
- is optional: $\omega(f) = 0$.
- is a leaf feature: $\lambda(f) = \langle 0..0 \rangle$.

Root r is decomposed into optional features (i.e., $\lambda(r) = \langle |P|..|P| \rangle$). Moreover, we add only one complex constraint representing the product line in disjunctive normal form such that $[\![m]\!] = \pi$:

$$\Psi = \{ \bigvee_{p \in \pi} (\bigwedge_{f \in p} f \wedge \bigwedge_{f \in P \setminus p} \neg f)\}.$$

Hence, $\mathcal{L}_{\mathcal{M}}$ is expressively complete. □

Complex constraints offer a strong and concise mechanism for documenting feature dependencies in a feature model. However, it is unclear how existing approaches in product-line research should be extended to integrate them. For example, in our survey (cf. Table 1) we looked at algorithms for optimal feature selection. Some of these approaches are based on genetic algorithms [6, 19]. There exists thus a catalog on how decomposition groups and cross-tree constraints are encoded into chromosomes of individuals. This is less challenging for simple constraints, since there are only four dependencies between two features: either one feature requires another feature (and vice versa), both are mutually exclusive, or there is no dependency. For arbitrary propositional formulas, however, it may be a considerable amount of extra work to modify these algorithms, and it may also be questionable whether a modification leads to an acceptable performance.

Another relevant aspect with an impact on existing product-line research is the interoperability of tools and tool reuse in general. Research could profit from a plug-and-play manner to combine existing and new concepts and tools. For this vision of incorporating different product-line tools, it is necessary that their feature model languages are translatable into each other, which gives rise to a product-preserving feature model transformation.

3.2 Relaxed Feature Models

In this section, we consider an alternative feature modeling language that is (1) syntactically very close to the language of basic feature models, (2) uses only simple constraints, and (3) is expressively complete. This language serves as a bridge between basic feature models and feature models using complex constraints. The difference to basic feature models is that relaxed feature models allow abstract features to be leaf features and to be part of simple constraints.

Definition 3.3. A *relaxed feature model* is defined as a 7-tuple $(N, P, r, \omega, \lambda, \Pi, \Psi)$ where

- N, P, r, ω, Π follow Definition 2.1,
- leaf features may also be abstract, and
- abstract features in simple constraints are allowed: $\Psi \subset \{f \Rightarrow g, f \Rightarrow \neg g \mid f, g \in N\}$.

The set of all relaxed feature models is denoted by $\mathcal{L}_{\mathcal{R}}$.

Our semantics also applies to $\mathcal{L}_{\mathcal{R}}$. With the following theorem, we show that those subtle changes already guarantee $\mathcal{L}_{\mathcal{R}}$ to be expressively complete.

Theorem 3.4. *Language $\mathcal{L}_{\mathcal{R}}$ is expressively complete.*

Proof. Let $\pi = \{p_1, ..., p_n\}$ be a product line with $\pi \in \mathcal{D}$. We construct a feature model $m = (N, P, r, \omega, \lambda, \Pi, \Psi)$ in $\mathcal{L}_R$ such that the following conditions hold.

$$\begin{array}{lcl}
P & = & \bigcup_{i=1}^{n} p_i \\
N & = & \{r, G, g_1, ..., g_n\} \dot\cup P \\
\omega(f) & = & \begin{cases} 0 & \text{if } f \notin \{r, G\} \\ 1 & \text{if } f \in \{r, G\} \end{cases} \\
\lambda(f) & = & \begin{cases} \langle |P|+1..|P|+1 \rangle & \text{if } f = r \\ \langle 1..1 \rangle & \text{if } f = G \\ \langle 0..0 \rangle & \text{otherwise} \end{cases} \\
\Pi & = & \{(r, f) \mid f \in P \cup \{G\}\} \cup \{(G, f) \mid f \in \{g_1, ..., g_n\}\} \\
\Psi & = & \bigcup_{i=1}^{n} \{g_i \Rightarrow f \mid f \in p_i\} \cup \{g_i \Rightarrow \neg f \mid f \in P \setminus p_i\}
\end{array}$$

Feature G is a mandatory abstract feature decomposed into an alternative-group with parent r and abstract sub-features $g_1, ..., g_n$. All concrete features in P are optional sub-features of r. Each sub-feature of G corresponds to one and only one product in π. For each feature in product p_i, we create a single requires constraint (i.e., $\forall f \in p_i : (g_i \Rightarrow f) \in \Psi$). For every other feature, we create a single excludes constraint (i.e., $\forall f \in P \setminus p_i : (g_i \Rightarrow \neg f) \in \Psi$). Each abstract leaf feature in the alternative-group now denotes a product in the product line π, such that $[\![m]\!] = \pi$ holds. Hence, $\mathcal{L}_R$ is expressively complete. □

In summary, feature models with complex constraints are expressively complete, but simplified assumptions in product-line research limit applicability of such feature models. Since $\mathcal{L}_{\mathcal{M}}$ and $\mathcal{L}_{\mathcal{R}}$ are equally expressive, a transformation from one language to another exists. Even more, semantic function $[\![.]\!]$ may map syntactically different feature models in $\mathcal{L}_{\mathcal{R}}$ to the same product line (e.g., if an abstract leaf feature is added). Hence, there may even exist more than one transformation.

Nevertheless, an acceptable transformation for us must comply with certain criteria. In particular, our goal is to find a transformation that does not degenerate the initial feature model hierarchy, since a different hierarchy without conserving present domain knowledge may cause confusion or might even be unusable as soon as the user starts manually working with the feature model to inspect analysis results. The construction proof of Theorem 3.4 invalidates this requirement. For this reason, we must think of a different transformation from $\mathcal{L}_{\mathcal{M}}$ to $\mathcal{L}_{\mathcal{R}}$.

4 ELIMINATING COMPLEX CONSTRAINTS

In this section, we present a transformation from feature models in $\mathcal{L}_{\mathcal{M}}$ to relaxed feature models in $\mathcal{L}_{\mathcal{R}}$. Our assumption is that relaxed feature models can be used in numerous application domains as a replacement for basic feature models. Hence, this translation is a potential compromise for feature models with complex constraints to be applicable for tools and approaches in product-line research only dealing with simple constraints.

First, we explain our algorithm for translating complex constraints into additional abstract features and simple constraints, while preserving the product line. Second, we give instructions on how further concepts (e.g., mutex-groups) can be resolved for our transformation algorithm to become generally applicable.

4.1 Translation to a Relaxed Feature Model

Not all complex constraints are of the same kind. Some complex constraints can be translated to an equivalent conjunction of simple constraints. For example, the complex constraint $f_1 \vee f_2 \Rightarrow f_3$ is equivalent to the conjunction of the simple constraints $f_1 \Rightarrow f_3$ and $f_2 \Rightarrow f_3$. To this end, we classify complex constraints further into two disjoint categories: *pseudo-complex* and *strict-complex* constraints. Pseudo-complex constraints are convertible to a set of simple constraints, whereas strict-complex constraints are not. More formally, a *pseudo-complex constraint* is a complex constraint ψ such that its conjunctive normal form has the form $\psi^{cnf} = \bigwedge c_i$ where $c_i \equiv (\neg f_1 \vee f_2)$ or $c_i \equiv (\neg f_1 \vee \neg f_2)$ for arbitrary features $f_1, f_2 \in \mathcal{F}$. Otherwise, we say that ψ is *strict-complex*. In the remainder, we assume that pseudo-complex constraints are already resolved and use the terms complex and strict-complex interchangeably.

The idea for transforming a feature model from $\mathcal{L}_{\mathcal{M}}$ to $\mathcal{L}_{\mathcal{R}}$ is to translate complex constraints to additional abstract features and simple constraints without adding or removing program variants from the respective product line. Moreover, the original feature model hierarchy is still embedded into the new one. Before giving an algorithm, let us first introduce a construct that we refer to as *abstract tree*.

Definition 4.1. Let $m = (N, P, r, \omega, \lambda, \Pi, \Psi)$ be a feature model in $\mathcal{L}_{\mathcal{M}}$. An *abstract tree* for m is a pair $(\widetilde{m}, \Phi)$, where $\widetilde{m} = (\widetilde{N}, \emptyset, \widetilde{r}, \widetilde{\lambda}, \widetilde{\omega}, \widetilde{\Pi}, \emptyset)$ is a feature model in $\mathcal{L}_{\mathcal{R}}$ such that $N \cap \widetilde{N} = \emptyset$ and Φ is a set of simple constraints in propositional logic over $N \cup \widetilde{N}$.

We use abstract trees to eliminate complex constraints from feature models. The initial assumption is that we can transform any cross-tree constraint to an abstract tree such that the complex constraint is semantically equivalent to the abstract tree in a given feature model (i.e., they both restrict the same combinations of features that cannot be activated together). We then exploit them to substitute each complex constraint in a feature model from $\mathcal{L}_{\mathcal{M}}$ with a corresponding abstract tree. All abstract trees and original feature model without complex constraints are then composed together into an equivalent feature model in $\mathcal{L}_R$. The join operation introduces a new root feature $\overline{r}$, decomposing into the respective root features of the components. Since our algorithm works incrementally (i.e., eliminating complex constraints one by one), we must extend $\mathcal{L}_{\mathcal{M}}$ to a language $\mathcal{L}_{\mathcal{M}'}$ in which abstract features can be leaf features and also occur in cross-tree constraints.

Definition 4.2. Let $m = (N, P, r, \omega, \lambda, \Pi, \Psi) \in \mathcal{L}_{\mathcal{M}'}$ and $(\widetilde{m}, \Phi)$ an abstract tree with $\widetilde{m} = (\widetilde{N}, \emptyset, \widetilde{r}, \widetilde{\lambda}, \widetilde{\omega}, \widetilde{\Pi}, \emptyset)$ such that $\overline{r} \notin N \cup \widetilde{N}$. The *join of m and $(\widetilde{m}, \Phi)$* is defined by

$$m \bullet (\widetilde{m}, \Phi) = (N \cup \widetilde{N} \cup \{\overline{r}\}, P, \overline{r}, \overline{\lambda}, \overline{\omega}, \overline{\Pi}, \Psi \cup \Phi),$$

where $\overline{\lambda} = \{(\overline{r}, \langle 2..2 \rangle)\} \cup \lambda \cup \widetilde{\lambda}$, $\overline{\omega} = \{(r, 1)\} \cup \omega \cup \widetilde{\omega}$, and $\overline{\Pi} = \{(\overline{r}, r), (\overline{r}, \widetilde{r})\} \cup \Pi \cup \widetilde{\Pi}$.

Transformation to Abstract Trees. Let $m \in \mathcal{L}_{\mathcal{M}'}$ and ϕ be a (not necessarily complex) constraint of m. Without loss of generality, we assume that ϕ is in conjunctive normal form (CNF),

$$\phi = (l_1^1 \vee \ldots \vee l_k^1) \wedge C_2 \wedge \ldots \wedge C_n,$$

with clauses $C_1, C_2, \ldots, C_n$ and literal l_j^i is the j-th literal of the i-th clause, denoted by $l_j^i \in C_i$. $|C_i|$ denotes the number of literals occurring in clause C_i, e.g., $|C_1| = k$. A literal is called *negative literal* if it has the form $\neg f$ where f is a feature of m. Otherwise, it is a *positive literal*. In both cases, the literal is referencing feature f.

The abstract tree for ϕ with respect to m is denoted by $\mathcal{T}(m, \phi) = (\widetilde{m}_\phi, \Phi_\phi)$. We first exploit the syntactic structure of the formula, yielding $\widetilde{m}_\phi$ as follows. Root feature $\widetilde{r}$ is added to $\widetilde{m}_\phi$. For each clause C_i, we add an abstract feature C_i to $\widetilde{m}_\phi$, such that C_i is a mandatory sub-feature of $\widetilde{r}$, i.e., $r \prec C_i$ and $\widetilde{\omega}(C_i) = 1$. For each literal l_j^i, we add an optional feature l_j^i to $\widetilde{m}_\phi$ as sub-feature of C_i, e.g., $C_1 \prec l_1^1$. Each clause C_i decomposes into an or-group, i.e., $\widetilde{\lambda}(C_i) = \langle 1..|C_i| \rangle$. Every configuration of $\widetilde{m}_\phi$ contains at least the root feature $\widetilde{r}$, features $C_1, \ldots, C_n$, and for each clause C_i at least one literal contained in C_i.

As a last step, we integrate the type of the literals, positive or negative, into the constraint set Φ_ϕ, such that the abstract tree $\mathcal{T}(m, \phi)$ may substitute the constraint ϕ in m. Therefore, consider a positive literal l_j^i, being a reference to some feature f in m. A configuration of m respecting constraint ϕ such that l_j^i is evaluated to true, contains feature f. Whenever the abstract feature l_j^i is part of a configuration, f is part of the configuration. Hence, for every positive literal l_j^i with reference to some feature f, we add a requires

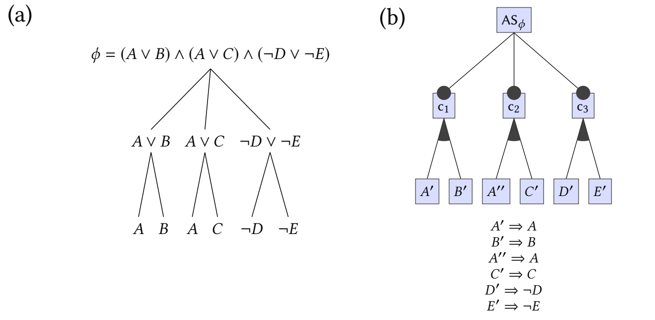

Figure 4: (a) a complex constraint in conjunctive normal form. (b) the corresponding abstract tree.

constraint (i.e., $l_j^i \Rightarrow f \in \Phi_\phi$). Conversely, for every negative literal l_j^i with reference to some feature f, we add an excludes constraint (i.e., $l_j^i \Rightarrow \neg f \in \Phi_\phi$). We illustrate the described procedure for a complex constraint and its respective abstract tree in Figure 4. Literals in the conjunctive normal form become primed features (i.e., for unique identification) and simple constraints in the abstract tree. The decomposition of the top feature is an and group and clauses become or-groups.

An abstract tree $\mathcal{T}(m, \phi)$ is capable of replacing the constraint ϕ in m without changing the semantics of the feature model. We denote by $m \setminus \phi$ the feature model m without constraint ϕ.

Lemma 4.3. *Let $m = (N, P, r, \omega, \lambda, \Pi, \Psi)$ be a feature model in $\mathcal{L}_{\mathcal{M}'}$ and $\phi \in \Psi$. Then $[\![m]\!] = [\![(m \setminus \phi) \bullet \mathcal{T}(m, \phi)]\!]$.*

Proof. Let $\mathcal{T}(m, \phi) = (\widetilde{m}_\phi, \Phi_\phi)$ with root feature $\overline{r}$.

$[\![m]\!] \subseteq [\![(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)]\!]$: Let $p \in [\![m]\!]$. Then there is a configuration $c \in C_m$ such that $c \cap P = p$ and $c \models \phi$. We construct a configuration $\widetilde{c} \in C_{(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)}$ such that $\widetilde{c} \cap P = p$. First, configuration c, root feature $\overline{r}$, and all abstract clause features $C_1, \ldots, C_n$ of ϕ are part of $\widetilde{c}$. For each feature $f \in c$, if there is a positive literal l_j^i in ϕ referencing f, add l_j^i to $\widetilde{c}$. For each feature $f \notin c$, if there is a negative literal l_j^i in ϕ referencing f, add l_j^i to $\widetilde{c}$. Since c and $\widetilde{c}$ at most differ in abstract features, $\widetilde{c} \cap P = p$.

It remains to be shown that $\widetilde{c} \in C_{(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)}$. Towards a contradiction, assume $\widetilde{c} \notin C_{(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)}$. Since $c \in C_m$ and $c \subseteq \widetilde{c}$, the contradiction arises from $(\widetilde{m}_\phi, \Phi_\phi)$ (i.e., (1) $\widetilde{m}_\phi$ or (2) Φ_ϕ). Suppose in case (1), there is a clause C_i with literals $l_1^i, \ldots, l_k^i \notin \widetilde{c}$. By construction, for each positive literal, the respective feature f is not part of $\widetilde{c}$ thus $f \notin c$. For each negative literal, the respective feature $f \in c \cap \widetilde{c}$. But this contradicts the assumption that $c \in C_m$ since $c \not\models \phi$, as clause C_i cannot be satisfied by c. In case (2), similar arguments apply.

$[\![(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)]\!] \subseteq [\![m]\!]$: Let $p \in [\![(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)]\!]$, i.e., there is a configuration $\widetilde{c}$ with $\widetilde{c} \cap P = p$. By $c = \widetilde{c} \cap N$ we obtain a candidate configuration with $c \cap P = p$. Proving $c \in C_m$ amounts to the reverse line of argumentation as above. □

Example 4.4. In Figure 5, we illustrate the elimination approach on the pizza product line extended by two complex constraints. All pseudo-complex constraints are translated to a set of simple constraints ① . All strict-complex constraints are translated into

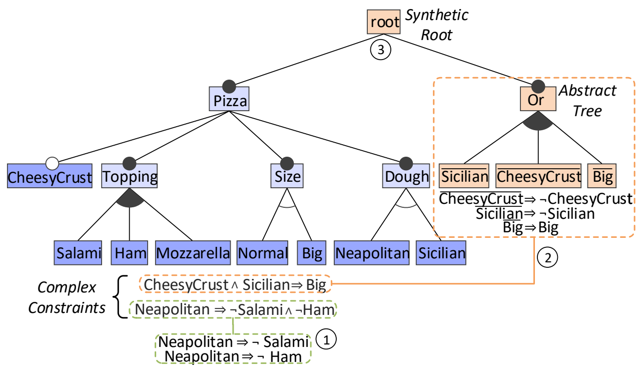

Figure 5: Example of eliminating complex constraints of the pizza product line.

abstract trees ②. Original feature model without complex constraints and abstract trees are composed together ③ to obtain a product-preserving feature model in $\mathcal{L}_\mathcal{R}$. The resulting feature model in Figure 5 now comprises five simple constraints and increased by five additional abstract features.

Total Correctness. Lemma 4.3 provides the desired elimination process. If the chosen constraint ϕ is a complex constraint in m, the resulting feature model $(m \setminus \phi) \bullet (\widetilde{m}_\phi, \Phi_\phi)$ has one complex constraint less than m. This is because the complex constraint ϕ is removed from m and only simple constraints from Φ_ϕ are added. Let $|m|_c$ denote the *number of complex constraints* in m. Furthermore observe that by Lemma 4.3, the tree structure of m is maintained during the elimination process, i.e., the m is structurally included in m', since only constraints are removed or added, and abstract trees are added to m. The following theorem shows how to incorporate Lemma 4.3 in an iterative elimination process, eventually obtaining a relaxed feature model from any feature model $m \in \mathcal{L}_\mathcal{M}$.

Theorem 4.5. *Let $m \in \mathcal{L}_\mathcal{M}$ be a feature model. Then there exists a feature model $m' \in \mathcal{L}_\mathcal{R}$ such that (1) the tree structure of m is embedded in that of m' (i.e., $\Pi_m \subseteq \Pi_{m'}$) and (2) $[\![m]\!] = [\![m']\!]$.*

Proof. Let $m \in \mathcal{L}_\mathcal{M}$ be a feature model with set of constraints Ψ. Set $m_0 = m$ with $\Psi_0 = \Psi$. Compute m_{i+1} from m_i as follows.

(1) Select complex constraint $\phi \in \Psi_i$ and
(2) set $m_{i+1} = (m_i \setminus c) \bullet \mathcal{T}(m_i, \phi)$.

Observe that for each $i \geq 1$, if $|\Psi_{i-1}|_c > 0$, then $|\Psi_i|_c = |\Psi_{i-1}|_c - 1$. Since Ψ is finite, say $|\Psi| = k$, there is an $n \leq k$ such that Ψ_n consists only of simple constraints. Since only complex constraints are removed from m_i and abstract trees are added in order to obtain m_{i+1}, the tree structure of m_i is included in that of m_{i+1}. By Lemma 4.3, the aforementioned observations, and transitivity of set equality (=), we get $m' = m_n \in \mathcal{L}_\mathcal{R}$ with (1) the tree structure of m included and (2) $[\![m]\!] = [\![m']\!]$. □

Given a constructive proof on the correctness of our algorithm, we are now able to overcome the limitations of basic feature models used in product-line research. However, our algorithm is based on the assumption that we already have a feature model in $\mathcal{L}_\mathcal{M}$, which is too restricting, since $\mathcal{L}_\mathcal{M}$ is not the only used language for real-world feature models. In the next section, we show how to make our algorithm applicable to four other common characteristics of feature modeling languages.

4.2 Translating Feature Model Dialects

Some feature modeling languages use additional concepts and decomposition groups in their concrete syntax to the ones we defined before. Thus, we propose a two-step algorithm that, first, transforms an arbitrary feature model to a feature model in $\mathcal{L}_\mathcal{M}$, and, second, transforms the resulting feature model to a relaxed feature model in $\mathcal{L}_\mathcal{R}$. The following described transformations are visualized in Figure 6.

Multiple Decomposition Types *($\mathcal{T}_\lambda$).* The language used by Czarnecki and Eisenecker [12] allows a feature to have multiple decompositions (e.g., an alternative- and an or-group below the same feature. To eliminate multiple groups $g_1, \ldots, g_n$ below a feature f, we set the features decomposition type to an and-group (i.e., $\lambda(f) = \langle n..n \rangle$), and substitute each group g_i by a mandatory abstract feature aux_i such that $f \prec aux_i$ and $aux_i \prec g_i$ for all $i = 1, \ldots, n$. Mandatory and optional features below f remain as-is.

Directed Acyclic Graphs *($\mathcal{T}_{DAG}$).* Some feature modeling languages, such as FORM [23] and FeatuRSEB [18], use directed acyclic graphs opposed to trees. If a feature g has multiple parents $f_1, \ldots, f_n$, we keep the relationship $f_1 \prec g$ and add an abstract feature aux_{i-1} for each $f_2, \ldots, f_n$ such that $f_i \prec aux_{i-1}$. Finally, we add constraints $g \Leftrightarrow aux_{i-1}$ for all $i = 2, \ldots, n$.

Group Cardinalities *($\mathcal{T}_{card}$).* There exist languages with custom group cardinalities [13]. If a feature g has a decomposition type different from the defined ones (e.g., $\lambda(g) = \langle a..b \rangle$), we set the decomposition type to an and-group (i.e., $\lambda(g) = \langle n..n \rangle$ with n being the number of sub-features of g) and add the following complex constraint:

$$g \Rightarrow \bigvee_{M \in P_{a,b}} \Big(\bigwedge_{f \in M} f \wedge \bigwedge_{f \in \{f' \mid g \prec f'\} \setminus M} \neg f \Big)$$

with $P_{a,b} = \{A \in 2^{\{f' \mid g \prec f'\}} \mid a \leq |A| \leq b\}$ being the set of all feature combinations of sub-features of g where each combination has at least a and at most b elements.

Mutex-Groups *($\mathcal{T}_{mutex}$).* Mutex-groups (i.e., groups where at most one feature can be selected) are another kind of prominent decomposition relations (e.g., in KConfig and CDL). If a feature f is a mutex-group decomposed into features $f_1, \ldots, f_n$, we change f's decomposition type to an and-group with one optional abstract sub-feature f'. Feature f' becomes an alternative-group with sub-features $f_1, \ldots, f_n$.

The presented transformations show that our approach is applicable to many other feature modeling languages. We can always develop a cascade of model transformations to eventually obtain a feature model in $\mathcal{L}_\mathcal{R}$. Correctness of transformations is omitted as it is much simpler compared to Section 4.1 and would require many further formalisms. Regarding the bigger picture, we are now in a position to investigate whether a mismatch between real-world feature models and product-line research exists by evaluating the usefulness of the transformed feature models.

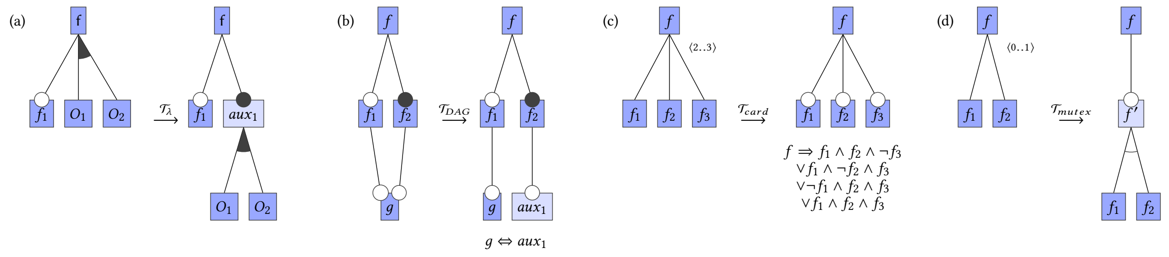
(a)
f
f_1
O_1
O_2
$\mathcal{T}_\lambda$
aux_1
(b)
$\mathcal{T}_{DAG}$
g
$g \Leftrightarrow aux_1$
(c)
⟨2..3⟩
$\mathcal{T}_{card}$
$f \Rightarrow f_1 \wedge f_2 \wedge \neg f_3 \vee f_1 \wedge \neg f_2 \wedge f_3 \vee \neg f_1 \wedge f_2 \wedge f_3 \vee f_1 \wedge f_2 \wedge f_3$
(d)
⟨0..1⟩
$\mathcal{T}_{mutex}$
f'

Figure 6: Graphical representation of a translation between concrete and abstract syntax: (a) adding an abstract feature to eliminate multiple decomposition types, (b) transforming a directed acyclic graph into a tree structure, (c) elimination of a custom group cardinality, and (d) elimination of a mutex-group.

5 EVALUATION AND DISCUSSIONS

We implemented a prototype in the open-source framework FeatureIDE and conducted experiments to evaluate the following research questions. Information on how to replicate the evaluation and where to find all data sets is given in the appendix.

RQ1 *What is the percentage of product lines representable by basic feature models?*

RQ2 *To what extent are simple and complex constraints used in real-world feature models?*

RQ3 *To what extent do feature models increase by transforming them to relaxed feature models?*

5.1 Open-Source Implementation

We implemented a prototype of our algorithm in FeatureIDE 3.1.0. The prototype allows to eliminate complex constraints of a feature model in FeatureIDE's own file format, resulting in an equivalent relaxed feature model.

Our elimination algorithm formulated in Section 4.1 relies on the conjunctive normal form of a constraint. The implementation also works with the *negation normal form* as-is, but the resulting abstract trees may constitute a different structure. Within the prototype, a user has the choice to either use the negation normal form, the conjunctive normal form, or the best for each constraint.

There is also the option to preserve the number of configurations. Our algorithm introduces new abstract features which do not increase the number of program variants, but may increase the number of configurations. If we add a bi-implication instead of a sole requires constraints (i.e., $f_1 \Leftrightarrow f_2$), we force a bijection between old and new configurations, which preserves the number of configurations. This is useful for applications that do not distinguish between configurations and program variants, but depend on their number (e.g., automated analyses or product-based sampling).

As an additional application scenario, our prototype forms the basis for general exporters to formats that only permit simple constraints (e.g., the FaMa file format [8]). Thereupon, we implemented an exporter to the basic FaMa file format [8], which is required as an input format for the BeTTy framework [34].

We use the prototypical implementation to answer research question *RQ3*. For our evaluation, we always use the combined approach to compute the best abstract tree for each complex constraint (i.e., resulting in the minimum amount of additional features and constraints). Moreover, we do not preserve the number of configurations, as it would only double the number of newly introduced require constraints.

5.2 Setup and Evaluated Feature Models

In our experiments, we are interested in large, industrial feature models and their cross-tree constraints. However, only few large feature models are publicly available, and online repositories, such as S.P.L.O.T. [28], mainly offer small toy examples that hardly reflect the complexity of real-world feature models. Hence, for our evaluation, we use four monthly snapshots of the automotive product line from our industrial partner with up to 18,616 features and 1,369 cross-tree constraints. Moreover, we evaluate our algorithm on variability models associated with two other variability modeling languages used in real software projects, namely KConfig and the *component definition language* (CDL).

KConfig was designed for the configuration management of the Linux kernel, but is also used in other software projects, such as *axTLS* or *CoreBoot*. CDL is specifically designed for the embedded system eCos. Each CDL model represents the configuration options for the eCos kernel for a specific hardware platform [9].

Since reference feature models are missing, we extended the CDLTools and LVAT developed by Berger et al. [9] to map the semantics of both languages to the FeatureIDE file format. As the semantics of all three languages are different, we had to make reasonable compromises. For all KConfig models, we neglected its tristate logic and assumed that features are either integrated in a program variant or not. In the mapping from KConfig and CDL to a feature model, we disregarded attributes (e.g., integer or strings) and removed cross-tree constraints that were either redundant (i.e., already covered by the hierarchy), unsatisfiable, or were referencing non-existent features. Overall, we analyzed four feature models from the automotive sector, 116 exported from CDL, and seven exported from KConfig.

5.3 Results and Discussion

***RQ1:* What is the percentage of product lines representable by basic feature models?** So far, we proved that basic feature models are expressively incomplete (cf. Theorem 2.6). However, the percentage of inexpressible feature models is still not identified. To this end, we decided to quantify the expressiveness of basic feature models (according to Definition 2.1) by implementing an algorithm that, given a number of concrete features, computes all valid basic feature models. We then calculated the number of distinct product lines covered by these feature models compared to the total number of possible product lines.[1]

[1]Given a subset $P \subseteq \mathcal{F}$ of concrete features, the total number of distinct product lines is $calc(P) = \sum_{k=0}^{|P|} \binom{|P|}{k}(-1)^k 2^{2^{|P|-k}}$. See https://oeis.org/A000371 for further information.

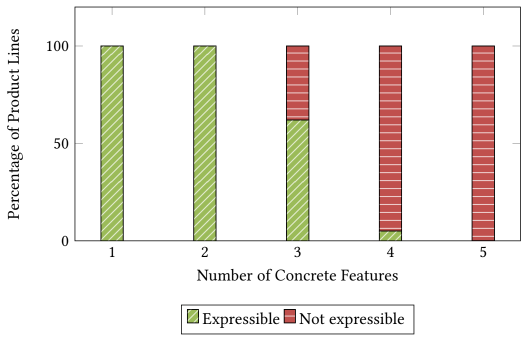

Figure 7: Percentage of product lines representable by basic feature models.

Figure 7 depicts the results for a maximum of five concrete features. Approximately 60% of all product lines covering three concrete features can be expressed with basic feature models. Three feature models covering inexpressible product lines are already exemplified in Figure 3. Less than 0.0005% of all available product lines can be expressed with basic feature models containing five concrete features. In theory, the likelihood that a desired product line is not *exactly* covered by a basic feature model is thus surprisingly high.

***RQ2:* To what extent are simple and complex constraints used in real-world feature models?** We investigate whether complex constraints are used in practice. We therefore analyzed the aforementioned real-world feature models for occurrences of pseudo- and strict-complex constraints. In Table 2, we summarize the results for our evaluated feature models. Since all 116 CDL models represent the eCos kernel targeted to different hardware platforms (i.e., slightly adapted), we only depict the minimum, maximum, and mean values. Based on the results, we can confidently conclude that complex constraints are heavily used in real-world feature models.

***RQ3:* To what extent do feature models increase by transforming them to relaxed feature models?** In Table 2, we depict the increase of features and constraints for all analyzed feature models after applying our transformation. In our automotive feature models, the maximum increase of features was measured with 2.58%, whereas the maximum increase of constraints was measured with 43.9%. The number of features of the Linux kernel increased by 582% and the number of constraints by 713%. Other KConfig models (e.g., EmbToolkit) also increased considerably in the number of new features and constraints. On average, the number of features for all 116 CDL models increased by 58%, whereas the number of constraints increased by 74%.

The increase in size depends on the complexity (i.e., the number of literals) of the complex constraints. We decided to also evaluate the number of literals per unprocessed strict-complex constraint and feature model. In Figure 8, we illustrate the results using box plots. Highlighting the complexity of some cross-tree constraints, we identified a constraint in the Linux kernel containing over 230 literals. On average, a complex constraint in the CDL models contains approximately five literals and a complex constraint in the automotive models contains approximately three literals.

The increase in size for all models ranges from below 1% to 1,403% in features and 7% to 4,648% in cross-tree constraints. Depending

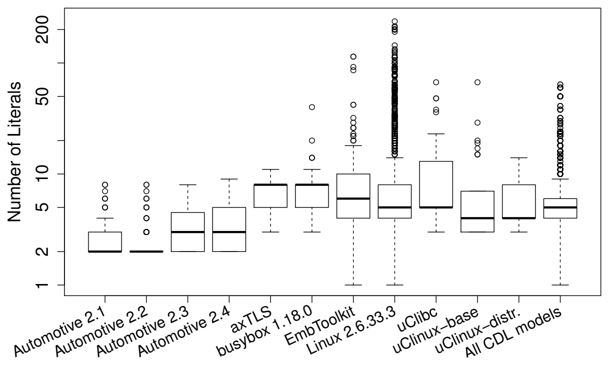

Figure 8: Number of literals of unprocessed complex constraints per evaluated feature model in logarithmic scale.

on the application, some models from CDL and KConfig could be particularly inefficient to process after the transformation to relaxed feature models is applied.

5.4 Threats to Validity

Internal Validity. We exported feature models from CDL and KConfig to the FeatureIDE file format. Both variability languages slightly differ in their semantics compared to the semantics of feature models (e.g., feature attributes). However, we exported the models according to the mapping concepts between CDL, KConfig, and feature modeling [9]. Moreover, other analysis projects, such as TypeChef [25], also translate KConfig to propositional logic, indicating that propositional logic is a common base. We also possibly removed cross-tree constraints (e.g., constraints referencing non-existent features), such that the resulting feature models may be even less complex than the original variability models.

Another threat is that our transformation does not minimize the number of additional features and constraints in general, as no logical minimization is performed. For instance, we may falsely classify a pseudo-complex constraint as strict-complex, leading to additional abstract trees. However, a manual inspection of our models revealed that these cases are rare.

External Validity. We evaluated 127 feature models in total, from which 116 models are based on CDL. That is, all these models are representing the eCos kernel adapted to different hardware platforms. We are aware that our results do not automatically transfer to other real-world feature models. Nevertheless, we used three different feature modeling languages and some of the largest publicly available product lines, which reflects that complex real-world feature models heavily rely on complex constraints.

6 DISCUSSIONS ON RELATED WORK

Formal Semantics of Feature Models. The idea of defining a general formal semantics to catch a variety of feature modeling dialects, and, thus, enhancing applicability of algorithms and research in general, is not new. Czarnecki et al. [13] proposed a cardinality-based notation to support their introduced concept of staged configurations. They proposed a similar formal semantics to the one we defined in this paper. Schobbens et al. [33] surveyed 12 feature modeling languages and based upon them proposed a general formal semantics called *Free Feature Model* [33]. They discussed properties such as expressive power, embeddability, and succinctness. All

Table 2: Overview of evaluated feature models including number of features and constraints before and after applying our constraint elimination approach.

		Automotive 2.1[1]	Automotive 2.2[1]	Automotive 2.3[1]	Automotive 2.4[1]	axTLS[2]	BusyBox 1.18[2]	EmbToolkit[2]	uClibc[2]	uClinux-base[2]	uClinux-dist[2]	Linux 2.6.33.3[2]	All CDL Models[3]
Size	Features	14,010	17,742	18,434	18,616	96	854	1,179	313	380	1,580	6,467	1,178 ≤ 1,259 ≤ 1,408
	Constraints	666	914	1,300	1369	14	123	323	56	3,455	197	3545	816 ≤ 877 ≤ 956
Complex Constraints	Strict	2.55%	3.93%	3.84%	5.62%	71.4%	73.9%	68.1%	64.2%	0.95%	21.3%	60.7%	9.00% ≤ 10.3% ≤ 12.0%
	Pseudo	14.8%	12.3%	15.2%	13.8%	21.4%	13.0%	10.5%	10.7%	0.00%	26.3%	32.3%	10.0% ≤ 11.2% ≤ 13.0%
	Sum	17.3%	16.2%	19.0%	19.4%	92.8%	86.9%	78.6%	74.9%	0.95%	47.6%	93.0%	19.0% ≤ 21.5% ≤ 25.0%
Increase	Features	0.62%	0.84%	1.29%	2.58%	115%	125%	1,403%	237%	85.5%	32.9%	582%	44.0% ≤ 58.0% ≤ 80.0%
	Constraints	25.0%	22.4%	37.3%	43.9%	621%	578%	4,648%	841%	7.46%	190%	713%	65.0% ≤ 74.0% ≤ 88.0%

[1] FeatureIDE model; [2] KConfig; [3] $Min \leq Mean \leq Max$ for all 116 CDL Models.

surveyed languages contain only simple constraints and the role of complex constraints is not discussed. Our work extends their prior theory and highlights the differences in expressive power with real numbers between basic feature models and feature models with complex constraints.

Eliminating Cross-Tree Constraints. Only little emphasis in the product-line community was put on the elimination of cross-tree constraints. Broek and Galvao [43] discussed the elimination of simple constraints by translating a feature model to a generalized feature tree, a structure allowing features to occur multiple times in different places including a potential negation. Our approach uses abstract features and simple constraints. Therefore, it is applicable for most tools requiring basic feature models as input. Gil et al. [17] aim to prove that all cross-tree constraints can be eliminated for the price of introducing a new set of features. Their approach is, however, only addressed from a theoretical perspective. Both approaches are beyond the goals we intend to accomplish, since some sort of constraints are typically supported in product-line research.

Consequences for Product-Line Research. Table 2 highlights the importance of complex constraints, since there was not a single evaluated feature model with only simple constraints. In the following, we also discuss consequences for some application domains.

The automated analysis deals with the computer-aided extraction of information from feature models (e.g., whether a feature is dead or how many products a feature model represents) [7]. To this end, common means include *SAT solvers* and *BDD solvers*, which use a propositional formula as input and check whether it is satisfiable. However, there exist also approaches based on description logic [29] or constraint satisfaction problems [49], for which it is unclear how complex constraints can be transformed. In such cases, our transformation can be applied, but analyses may take longer.

Some approaches for feature-model synthesis seem to be simpler to adapt. She et al. [37] use a propositional formula as input to automatically derive a basic feature model. They additionally carry a propositional rest in case the feature model is not equivalent to the input formula. Acher et al. [1] use product descriptions for synthesizing a basic feature model. They intentionally over-approximate the configurations, which can be prevented by additional complex constraints. In both cases, our algorithm can be used to eliminate the remaining constraints by converting them to simple constraints.

The problem of generating artificial feature models with complex constraints for evaluating the quality of analysis algorithms was proposed by Thüm et al. [40]. There exist also ETHOM as part of BeTTy [34], an algorithm for generating computationally hard feature models. Nevertheless, ETHOM is based on an evolutionary algorithm and generates only basic feature models. Our work makes aware that these are likely to be unrepresentative and generating computationally hard complex constraints is non-trivial.

The optimal selection of features with non-functional properties attached to them is either solved exactly (e.g., linear programming or constraint satisfaction solving) or heuristically (e.g., using evolutionary algorithms). To this end, feature models are transformed into other problems, for which algorithms and solutions exist. Usually, a catalog is presented on how to transform the parts of a feature model (i.e., decomposition relations and cross-tree constraints). If this catalog only covers simple constraints, our algorithm can be applied for feature models using complex constraints. However, as concluded before, scalability depends on the input feature model.

7 CONCLUSION

Various feature modeling languages exist to describe valid combinations of features in a software product line. We showed that numerous utilized languages in product-line research only use simple constraints, which we confirmed to be a too simplified assumption for real-world feature models. We analyzed whether simple constraints are enough for feature modeling and proposed an algorithm to eliminate complex constraints. Our conducted experiments show that the algorithm leads to significantly increased feature models.

For large feature models, our algorithm may render feature model applications infeasible, but the elimination of complex constraints is irrefutable for practical product-line engineering: researchers and practitioners can more easily reuse tools and some research gets easier applicable to real-world problems. Given our algorithm, simple constraints are sufficient if (a) users do not need to inspect the intermediate representation and (b) if scalability with more features or constraints does not pose any problems.

Nevertheless, we advocate that product-line research should consider complex constraints as default in the future. We further think that a community effort is needed to evaluate which and how approaches tailored to basic feature models can be applied to complex constraints. In conclusion, complex constraints are heavily used in real-world feature models. Research on product lines should either include them or at least discuss consequences of their elimination, if feasible at all.

A APPENDIX: REPLICATION PACKAGE

We provide access to 127 large real-world feature models with thousands of features and cross-tree constraints in the FeatureIDE file format that can be easily exported to other feature modeling dialects (e.g., SXFM). These feature models can be used in future research in different analysis contexts. Furthermore, we provide the source code and software artifacts necessary to translate a feature model with complex constraints to a relaxed feature model, and to reproduce our experimental results. The package is self-contained such that all empirical results can be reproduced automatically.

In total, our replication package contains 123 feature models in the FeatureIDE file format translated from KConfig and CDL, as well as 4 obfuscated feature models from our industry partner. Furthermore, we provide two Java Eclipse projects and one Scala project. The first Eclipse project is for calculating the expressive power of non-equivalent feature models with only simple constraints. The second Eclipse project is for analyzing our constraint elimination approach empirically and to generate all statistics. For the statistics, we rely on our constraint elimination algorithm, which we integrated into FeatureIDE 3.1.0[2] and later versions. The Scala project is an extended version of the *Linux Variability Analysis Project*,[3] offering an exporter from KConfig to FeatureIDE.

The objective of the provided artifacts is to enable the analysis of feature models with respect to their cross-tree constraints. We offer possibilities to translate a feature model with complex constraints to a relaxed feature model (i.e., a basic feature model with abstract features). Furthermore, the data set can be used for future evaluations by other researchers. Our complex-constraint elimination algorithm is integrated into the widely-used and long-living tool FeatureIDE, bridging the gap between expressively complete and lesser expressive feature modeling languages. For instance, FaMa is a basic feature modeling file format. The FaMa exporter of FeatureIDE internally uses our algorithm to eliminate complex constraints if necessary.

Download and Setup. To ensure easy access for replication of our experiments, we created a publicly available repository on GitHub.[4] Compulsory for executing the programs is a Java 7 compiler and possibly also a Scala compiler with a version higher than 2.11. The Java projects are initially prepared to be used with Eclipse. Other than that, the repository is self-contained so that users can readily download all artifacts and run the experiments. The repository inherits a detailed documentation explaining how to setup and run the tools.

Source Code and Data. Our experimental results have been successfully evaluated by the Artifact Evaluation Committee and considered to be *reusable*. The repository consists of four major parts.

(1) **Analyzer for the Expressive Power of Basic Feature Models.** This tool is used in Section 5.3 to calculate the number of theoretically possible product lines that a basic feature modeling languages can express. It takes a number of concrete features as input and computes all variations of a basic feature model. It then counts the number of non-equivalent product lines that are represented by these feature models.
(2) **Experimental Evaluation.** All experimental results explained in Section 5 can be calculated by this project. The user specifies a list of feature models (or a folder containing feature models) as input and the project generates all statistics (i.e., number of complex constraints, increase in size after translation, and complexity of all complex constraints).
(3) **KConfig Translator.** The KConfig translator is an extended version of the Linux Variability Analysis Project developed by Berger et al. [9]. A user specifies a model in the `exconfig` file format (an extended KConfig file format) and receives an approximation of the feature model in the FeatureIDE file format.
(4) **Large Real-World Feature Models.** All 127 extracted and evaluated feature models are provided in the FeatureIDE file format, both in its complex and relaxed version.

FeatureIDE Integration. Our algorithm described in Section 4 is integrated into FeatureIDE and is already usable in practice (e.g., exporting to basic feature modeling formats). The context menu of the `model.xml` provides the option to translate that feature model into a product-equivalent feature model with only simple constraints. The appearing dialog provides options to (1) chose a strategy for all abstract trees (*conjunctive normal form*, *negation normal form*, or a combination of both), (2) preserve the number of configurations leading to a doubling in the number of additional require constraints, and (3) to additionally remove redundant constraints that may arise in the process.

ACKNOWLEDGMENTS

We gratefully acknowledge fruitful discussions on the expressive power of feature models with Arthur Hammer, Malte Lochau, Christian Kästner, Reimar Schröter, and Gunter Saake. We thank Thorsten Berger for his support in retrieving feature diagrams for real-world feature models. Thanks to Niklas Lehnfeld for his help with developing the software artifacts. This work was supported by the European Union within project HyVar (grant agreement H2020-644298), the DFG (German Research Foundation) under the Priority Programme SPP1593, and NSF grant 1552944.

REFERENCES

[1] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. On Extracting Feature Models From Product Descriptions. In *VaMoS*. ACM, New York, NY, USA, 45–54. DOI: http://dx.doi.org/10.1145/2110147.2110153
[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2011. Managing Feature Models With Familiar: a Demonstration of the Language and its Tool Support. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems.* ACM, 91–96.
[3] Ra'Fat Al-Msie 'deen, Marianne Huchard, Abdelhak-Djamel Seriai, Christelle Urtado, and Sylvain Vauttier. 2014. Reverse Engineering Feature Models from Software Configurations Using Formal Concept Analysis. In *CLA 2014: Eleventh International Conference on Concept Lattices and Their Applications (CEUR-Workshop)*, Sebastian Rudolph Karell Bertet (Ed.), Vol. 1252. Ondrej Krídlo, Košice, Slovakia, 95–106. https://hal-auf.archives-ouvertes.fr/hal-01075524
[4] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. Clafer: Unifying Class and Feature Modeling. *Software & Systems Modeling* (2013), 1–35.
[5] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 7–20.

[2] https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.1.0
[3] https://code.google.com/archive/p/linux-variability-analysis-tools/
[4] https://github.com/AlexanderKnueppel/is-there-a-mismatch

[6] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2005. Automated Reasoning on Feature Models. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE).* 491–503.

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

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

[9] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. 2010. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoSâĂŽ10), Linz, Austria, January.* 27–29.

[11] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning About Product-line Evolution Using Complex Feature Model Differences. *Automated Software Engineering* 23, 4 (2016), 687–733.

[12] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM/Addison-Wesley, New York, NY, USA.

[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-Based Feature Models and Their Specialization. *Software Process: Improvement and Practice* 10 (2005), 7–29.

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

[15] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. 2012. Evolutionary Search-based Test Generation for Software Product Line Feature Models. In *Advanced Information Systems Engineering.* Springer, 613–628.

[16] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. 2005. The PLUSS Approach: Domain Modeling with Features, Use Cases and Use Case Realizations. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05).* Springer-Verlag, Berlin, Heidelberg, 33–44. DOI: http://dx.doi.org/10.1007/11554844_5

[17] Yossi Gil, Shiri Kremer-Davidson, and Itay Maman. 2010. Sans Constraints? Feature Diagrams vs. Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC).* Springer, Berlin, Heidelberg, 271–285.

[18] M. L. Griss, J. Favaro, and M. d' Alessandro. 1998. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98).* IEEE Computer Society, Washington, DC, USA, 76–. http://dl.acm.org/citation.cfm?id=551789.853486

[19] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *J. Syst. Softw.* 84, 12 (Dec. 2011), 2208–2221. DOI: http://dx.doi.org/10.1016/j.jss.2011.06.026

[20] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01).* IEEE Computer Society, Washington, DC, USA, 45–. DOI: http://dx.doi.org/10.1109/WICSA.2001.948406

[21] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2013. On Extracting Feature Models from Sets of Valid Feature Combinations. In *International Conference on Fundamental Approaches to Software Engineering.* Springer, 53–67.

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

[23] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1 (Jan. 1998), 143–168.

[24] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE).* IEEE, Washington, DC, USA, 611–614. Formal demonstration paper.

[25] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD).* ACM, New York, NY, USA, 25–32. DOI: http://dx.doi.org/10.1145/1868688.1868693

[26] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Feature Model Synthesis with Genetic Programming. In *International Symposium on Search Based Software Engineering.* Springer, 153–167.

[27] Roberto E Lopez-Herrejon, Lukas Linsbauer, José A Galindo, José A Parejo, David Benavides, Sergio Segura, and Alexander Egyed. 2015. An Assessment of Search-based Techniques for Reverse Engineering Feature Models. *Journal of Systems and Software* 103 (2015), 353–369.

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

[29] Mahdi Noorian, Alireza Ensan, Ebrahim Bagheri, Harold Boley, and Yevgen Biletskiy. 2011. Feature Model Debugging Based on Description Logic Reasoning.. In *DMS*, Vol. 11. Citeseer, 158–164.

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

[31] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. In *Proc. World Conf. Integrated Design and Process Technology (IDPT).*

[32] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *VaMoS.* ACM, NY, 11–22.

[33] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007), 456–479.

[34] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, New York, NY, USA, 63–71. DOI: http://dx.doi.org/10.1145/2110147.2110155

[35] Sergio Segura, José A Parejo, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. 2014. Automated Generation of Computationally Hard Feature Models Using Evolutionary Algorithms. *Expert Systems with Applications* 41, 8 (2014), 3975–3992.

[36] Hazim Shatnawi and H Conrad Cunningham. 2017. Mapping SPL Feature Models to a Relational Database. In *Proceedings of the SouthEast Conference.* ACM, 42–49.

[37] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE).* ACM, New York, NY, USA, 461–470. DOI: http://dx.doi.org/10.1145/1985793.1985856

[38] Jiangfan Shi, Myra B Cohen, and Matthew B Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Fundamental Approaches to Software Engineering.* Springer, 270–284.

[39] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. 2014. On the Use of Feature-Oriented Programming for Evolving Software Product Lines — A Comparative Study. *Science of Computer Programming (SCP)* 93, A (2014), 65 – 85. DOI: http://dx.doi.org/10.1016/j.scico.2013.10.010

[40] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE).* IEEE, Washington, DC, USA, 254–264.

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

[42] C. Reid Turner, Alexander L. Wolf, Alfonso Fuggetta, and Luigi Lavazza. 1998. Feature Engineering. In *Proc. Int'l Workshop Software Specification and Design (IWSSD).* IEEE, Washington, DC, USA, 162–164.

[43] PM van den Broek and I Galvao Lourenco da Silva. 2009. Analysis of feature models using generalised feature trees. (2009).

[44] Arie van Deursen and Paul Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Computing and Information Technology* 10, 1 (2002), 1–17.

[45] Shuai Wang, Shaukat Ali, Arnaud Gotlieb, and Marius Liaaen. 2017. Automated Product Line Test Case Selection: Industrial Case Study and Controlled Experiment. *Software and Systems Modeling (SoSyM)* 16, 2 (2017), 417–441.

[46] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. 2016. Mind the Gap! Automated Anomaly Detection for Potentially Unbounded Cardinality-Based Feature Models. In *International Conference on Fundamental Approaches to Software Engineering.* Springer, 158–175.

[47] Jules White, Brian Dougherty, and Douglas C Schmidt. 2009. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software* 82, 8 (2009), 1268–1284.

[48] Jules White, José A Galindo, Tripti Saxena, Brian Dougherty, David Benavides, and Douglas C. Schmidt. 2014. Evolving Feature Model Configurations in Software Product Lines. *J. Systems and Software (JSS)* 87, 0 (2014), 119–136.

[49] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2008. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC).* IEEE, Washington, DC, USA, 225–234.