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

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

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

Stephan Mennicke  
TU Braunschweig, Germany  
mennicke@ovgu.de

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

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

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 acquired 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 research 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 evalua-
tions [19, 35], and optimal feature selection [6, 19, 47, 48]. More
surprisingly, the number of annually proposed methods that are based
on expressively incomplete feature modeling languages does not
seem to decrease over time, as we still identified several publi-
cations 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 arbi-
trary 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 Fea-
tureIDE [24], Familiar [2], or Clafer [4]. Other variability lan-
guages, such as KConfig and CDL, where feature model approxima-
tions 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 dif-
ferent stages in the engineering process, a feature model transfor-
mation is necessary. However, feature models with complex con-
straints cannot generally be transformed into ones with only simple
constraints, as their languages differ in expressive power. Neverthe-
less, 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 re-
search. However, relaxed feature models may increase significantly
in the number of features and constraints. We analyze the usefulness
of this transformation on real-world feature models. In particu-
lar, 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 lan-
guages 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 pre-
dominantly used in product-line research. Thereupon, we inves-
tigate its expressive power. A basic feature model is a hierarchi-
cally 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 ex-
clude constraints can be specified [12, 22]. In Figure 1, we exem-
ply a basic feature model representing a product line for pizzas.

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 ei-
ther 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 \( \Rightarrow \) 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 con-
crete syntax (i.e., as we did above), or formally by defining a for-
mal model. We are interested in describing our approach formally
to precisely and unambiguously reason about our aforementioned
contributions. For this purpose, Schobbens et al. [33] proposed a
generic formal semantics to catch a variety of older feature model-
ing dialects. To increase expressiveness, they used directed acyclic
graphs instead of trees. However, in our experience, the vast major-
ity of feature modeling languages build upon a tree structure. Thus,
we give a modified version of the semantics for basic feature mod-
els compared to the one Schobbens et al. proposed [33]. More pre-
cisely, 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 anal-
ysis 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
digraph, 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 fea-
ture models for five application domains.

<table>
<thead>
<tr>
<th>Research Area</th>
<th>Proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of feature models</td>
<td>[11, 34, 46]</td>
</tr>
<tr>
<td>Feature model synthesis</td>
<td>[3, 21, 26, 27, 37]</td>
</tr>
<tr>
<td>Generating artificial feature models</td>
<td>[19, 35]</td>
</tr>
<tr>
<td>Product-line testing</td>
<td>[15, 38]</td>
</tr>
<tr>
<td>Optimal feature selection</td>
<td>[6, 19, 47, 48]</td>
</tr>
</tbody>
</table>
Is There a Mismatch between Real-World Feature Models and Product-Line Research?

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 \( \{ 12 \} \) (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 interchangeably 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.
- \( \omega : N \rightarrow \{ 0, 1 \} \) is a function labeling a feature as either optional (0) or mandatory (1) with \( \omega(r) = 1 \).
- \( \lambda : N \rightarrow \mathbb{N} \times \mathbb{N} \) 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 \((1..)\) for alternative-groups, \((1..n)\) for or-groups with \( n \) sub-features, \((n..)\) for and-groups with \( n \) sub-features, and \((0..)\) for leaf features.
- \( \Pi \subseteq N \times N \) is a decomposition relation. We denote \((f, g) \in \Pi\) as \( f < g \), 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., \( \forall f \in N, \omega(f) = (0, 0) \) then \( f \in P \)). Moreover, only features of an and-group can be mandatory (i.e., for \( f, g \in N \), \( g \in f \) and \( \omega(f) = 1 \), then \( \lambda(g) = (n..) \) with \( n \) being the number of sub-features of \( g \)).

**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.

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

![Figure 2: Reduced feature model for the pizza product line.](image)

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 \( c \models_m m \), if and only if
- it contains the root feature: \( r \in c \).
- it satisfies the decomposition type: \( \forall f \in c, \lambda(f) = (0, 0) \).
- it satisfies each cross-tree constraint: \( \forall \psi \in \Psi : \bigwedge_{f \in \text{Concrete } N} (\neg f \Rightarrow \psi) \).

We denote the set of all valid configurations of \( m \) by \( \mathcal{C}_m \).

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 \( \mathcal{D} = 2^\mathcal{P} \) 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 \( \mathcal{M}_B := \{ c \cap P \mid c \in \mathcal{C}_m \} \).

**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.

\[ [m]_B = \{\{N, Sa\}, \{N, H\}, \{N, M\}, \{N, Sa, H\}, \{N, Sa, M\}, \{N, H, M\}, \{N, Sa, H, M\}, \{B, M\}, \{B, Sa, M\}, \{B, H, M\}, \{B, Sa, H, M\}\} \]

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 \textit{expressively complete}, if the domain of the language equals \(\mathcal{D}\) and \textit{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., \(\mathcal{D}_B \neq \mathcal{D}\)).

**Proof.** It is sufficient to only show one product line \(\pi\) 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, \(\{\{f_1\}, \{f_2\}\}\) must hold. This, however, is contradicting to the product line \(\pi\). 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.

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 \(\mathcal{L}_B\) 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 \([\cdot]_B\) (cf. Definition 2.4) carries over to both languages. For convenience, we simply use \([\cdot]_B\) in the following.

We define the syntactic domain \(\mathcal{L}_M\) 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 \(N, \text{i.e.}, \Psi \subseteq \mathcal{B}(N)\).

**Theorem 3.2.** The language \(\mathcal{L}_M\) 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 \(m \in \mathcal{L}_M\) with \(P = \bigcup_{p \in \pi} p\) and each feature \(f \in P\) holds the following conditions.

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

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

\[ \Psi = \{ \bigvee_{p \in \pi} \bigwedge_{f \in p} f \land \bigwedge_{f \in \mathcal{V}_p} \neg f \} \]

Hence, \(\mathcal{L}_M\) 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
Is There a Mismatch between Real-World Feature Models and Product-Line Research?

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

- \(N, P, r, \omega, \Pi\) 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 \(L_R\).

Our semantics also applies to \(L_R\). With the following theorem, we show that those subtle changes already guarantee \(L_R\) to be expressively complete.

Theorem 3.4. Language \(L_R\) is expressively complete.

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

\[
P = \bigcup_{i=1}^{n} p_i
\]

\[
N = \{r, G, g_1, \ldots, g_n\} \cup P
\]

\[
\omega(f) = \begin{cases} 0 & \text{if } f \notin \{r, G\} \\ 1 & \text{if } f \in \{r, G\} \\ 1 & \text{if } f \in \{r, G\} \\ (|P| + 1) & \text{if } f = r \\ (1) & \text{if } f = G \\ (0..0) & \text{otherwise} \\ \end{cases}
\]

\[
\lambda(f) = \begin{cases} (1..1) & \text{if } f = G \\ (0..0) & \text{otherwise} \\ \end{cases}
\]

\[
\Pi = \bigcup_{i=1}^{n} \{g_i \Rightarrow f \mid f \in P \cup \{G\}\} \cup \{(G, f) \mid f \in \{g_1, \ldots, g_n\}\}
\]

\[
\Psi = \{ g_i \Rightarrow f \mid f \in p_i \} \cup \{ g_i \Rightarrow \neg f \mid f \in P \setminus p_i \}
\]

Feature \(G\) is a mandatory abstract feature decomposed into an alternative-group with parent \(r\) and abstract sub-features \(g_1, \ldots, g_n\). All concrete features in \(P\) are optional sub-features of \(r\). Each sub-feature of \(G\) corresponds to one and only one product in \(\pi\). 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 \(\pi\), such that \([m] = \pi\) holds. Hence, \(L_R\) 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 \(L_M\) and \(L_R\) are equally expressive, a transformation from one language to another exists. Even more, semantic function \([\cdot]\) may map syntactically different feature models in \(L_R\) 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 validates this requirement. For this reason, we must think of a different transformation from \(L_M\) to \(L_R\).

4 ELIMINATING COMPLEX CONSTRAINTS

In this section, we present a transformation from feature models in \(L_M\) to relaxed feature models in \(L_R\). 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 \(f_1 \lor f_2 \Rightarrow f_3\) 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 \(\psi\) such that its conjunctive normal form has the form \(\psi_{cnf} = \land c_i\) where \(c_1 \equiv (\neg f_1 \lor f_2)\) or \(c_1 \equiv (\neg f_1 \lor \neg f_2)\) for arbitrary features \(f_1, f_2 \in F\). Otherwise, we say that \(\psi\) 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 \(L_M\) to \(L_R\) 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 \( \mathcal{L}_M \). An abstract tree for \( m \) is a pair \((\overline{m}, \Phi)\), where \( \overline{m} = (\overline{N}, \overline{0}, \overline{r}, \overline{\lambda}, \overline{\omega}, \overline{\Pi}, \overline{\Psi}) \) is a feature model in \( \mathcal{L}_R \) such that \( N \cap \overline{N} = \emptyset \) and \( \Phi \) is a set of simple constraints in propositional logic over \( N \cup \overline{N} \).

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 \( \mathcal{L}_M \) 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 \( \mathcal{L}_M \) to a language \( \mathcal{L}_M' \), in which abstract features can be leaf features and also occur in cross-tree constraints.

Definition 4.2. Let \( m = (N, P, r, \omega, \lambda, \Pi, \Psi) \in \mathcal{L}_M \) and \((\overline{m}, \Phi)\) an abstract tree with \( \overline{m} = (\overline{N}, \overline{0}, \overline{r}, \overline{\lambda}, \overline{\omega}, \overline{\Pi}, \overline{\Psi}) \) such that \( \overline{r} \notin N \cup \overline{N} \). The join of \( m \) and \( \overline{m} \) is defined by

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

where \( \overline{\lambda} = \{(\overline{r}, (r_1, \ldots, r_m)) \mid \lambda \cup \overline{\lambda} = \{(r_1, 1)\} \cup \omega \cup \overline{\omega}, \) and \( \overline{\Pi} = \{(\overline{r}, \overline{r}), (\overline{r}, \overline{\overline{r}}), \overline{\overline{r}} \} \cup \overline{\Pi} \).

Transformation to Abstract Trees. Let \( m \in \mathcal{L}_M \) be an abstract tree, \( \phi \) is a (not necessarily complex) constraint of \( m \). Without loss of generality, we assume that \( \phi \) is in conjunctive normal form (CNF),

\[
\phi = (l_{i1} \lor \cdots \lor l_{ik})(\land C_1 \land C_2 \land \cdots \land C_n).\]

with clauses \( C_1, C_2, \ldots, C_n \) and literal \( l_j \) is the \( j \)-th literal of the \( i \)-th clause, denoted by \( l_j \in C_i \), \( |C_i| \) 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 \( \phi \) with respect to \( m \) is denoted by \( T(m, \phi) = (\overline{m}, \Phi) \). We first exploit the syntactic structure of the formula, yielding \( \overline{m} \) as follows. Root feature \( \overline{r} \) is added to \( \overline{m} \). For each clause \( C_i \), we add an abstract feature \( \overline{C}_i \) to \( m \), such that \( \overline{C}_i \) is a mandatory sub-feature of \( \overline{r} \), i.e., \( r \prec \overline{C}_i \) and \( \overline{\omega}(\overline{C}_i) = 1 \). For each literal \( l_j \), we add an optional feature \( \overline{l}_j \) to \( \overline{m} \) as sub-feature of \( \overline{C}_i \), e.g., \( C_i \prec l_j \). Each clause \( C_i \) decomposes into an or-group, i.e., \( \overline{\lambda}(C_i) = \langle 1, \mid C_i \mid \rangle \). Every configuration of \( \overline{m} \) contains at least the root feature \( \overline{r} \), features \( \overline{C}_1, \ldots, \overline{C}_n \), 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 \( \Phi \), such that the abstract tree \( T(m, \phi) \) may substitute the constraint \( \phi \) in \( m \). Therefore, consider a positive literal \( l_j \), being a reference to some feature \( f \) in \( m \). A configuration of \( m \) respecting constraint \( \phi \) such that \( l_j \) is evaluated to true, contains feature \( f \). Whenever the abstract feature \( \overline{l}_j \) is part of a configuration, \( f \) is part of the configuration. Hence, for every positive literal \( l_j \) with reference to some feature \( f \), we add a requires constraint (i.e., \( l_j \Rightarrow f \in \Phi \)). Conversely, for every negative literal \( \overline{l}_j \) with reference to some feature \( f \), we add an excludes constraint (i.e., \( \overline{l}_j \Rightarrow \neg f \in \Phi \)). 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 \( T(m, \phi) \) is capable of replacing the constraint \( \phi \) in \( m \) without changing the semantics of the feature model. We denote by \( m \setminus \phi \) the feature model \( m \) without constraint \( \phi \).

Lemma 4.3. Let \( m = (N, P, r, \omega, \lambda, \Pi, \Psi) \) be a feature model in \( \mathcal{L}_M \) and \( \phi \in \Psi \). Then \( [m] = [[m \setminus \phi] \bullet T(m, \phi)] \).

Proof. Let \( T(m, \phi) = (\overline{m}, \Phi) \) with root feature \( \overline{r} \).

\[
[m] \subseteq [(m \setminus \phi) \bullet (\overline{m}, \Phi)].
\]

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 \( \overline{c} \in C(\overline{m} \setminus \phi) \bullet (\overline{m}, \Phi) \) such that \( \overline{c} \cap P = p \). First, configuration \( c \), root feature \( \overline{r} \), and all abstract clause features \( \overline{C}_1, \ldots, \overline{C}_n \) of \( \phi \) are part of \( \overline{c} \). For each feature \( f \in c \), if there is a positive literal \( l_j \) in \( \phi \) referencing \( f \), add \( \overline{l}_j \) to \( \overline{c} \). For each feature \( f \notin c \), if there is a negative literal \( \overline{l}_j \) in \( \phi \) referencing \( f \), add \( \overline{l}_j \) to \( \overline{c} \). Since \( c \) and \( \overline{c} \) at most differ in abstract features, \( \overline{c} \cap P = p \).

It remains to be shown that \( \overline{c} \in C(m \setminus \phi) \bullet (\overline{m}, \Phi) \). Towards a contradiction, assume \( \overline{c} \notin C(m \setminus \phi) \bullet (\overline{m}, \Phi) \). Since \( c \in C_m \) and \( c \subseteq \overline{c} \), the contradiction arises from \( (\overline{m}, \Phi) \) (i.e., \( (\overline{m}, \Phi) \) or \( (\overline{m}, \Phi) \)). Suppose in case (1), there is a clause \( C_i \) with literals \( l_{i1}, \ldots, l_{ik} \notin \overline{c} \). By construction, for each positive literal, the respective feature \( f \) is not part of \( \overline{c} \) thus \( f \notin c \). For each negative literal, the respective feature \( f \in c \cap \overline{c} \). 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.

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

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
Lemma 4.3, the aforementioned observations, and transitivity of constraints and abstract trees are composed together to obtain a product-preserving feature model in \( L_R \). 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 \( \phi \) is a complex constraint in \( m \), the resulting feature model \( (m \setminus \phi) \) has one complex constraint less than \( m \). This is because the complex constraint \( \phi \) is removed from \( m \) and only simple constraints from \( \Phi_f \) 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 \( m \in L_M \).

**Theorem 4.5.** Let \( m \in L_M \) be a feature model. Then there exists a feature model \( m' \in L_R \) 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 \( m \in L_M \) be a feature model with set of constraints \( \Psi \). Set \( m_0 = m \) with \( \Psi_0 = \Psi \). Compute \( m_{i+1} \) from \( m_i \) as follows.

1. Select complex constraint \( \phi \in \Psi \) and \( \phi \notin \Psi_f \).
2. Set \( m_{i+1} = (m_i \setminus \phi) \bullet T(m_i, \phi) \).

Observe that for each \( i \geq 1 \), if \( |\Psi_{i-1}|_c > 0 \), then \( |\Psi| = |\Psi_i|_c = 1 \). Since \( \Psi \) is finite, say \( |\Psi| = k \), there is an \( n \leq k \) such that \( \Psi_f \) 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 \( m' = m_n \in L_R \) 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 \( L_M \), which is too restricting, since \( L_M \) 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 \( L_M \), and, second, transforms the resulting feature model to a relaxed feature model in \( L_R \). The following described transformations are visualized in Figure 6.

#### Multiple Decomposition Types (\( T_{\text{Dec}} \))

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 \( g_1, ..., g_n \) below a feature \( f \), we set the features decomposition type to an and-group (i.e., \( \lambda(f) = (n.n) \)), and substitute each group \( g_i \) by a mandatory abstract feature \( aux_i \) such that \( f < aux_i \) and \( aux_i < g_i \) for all \( i = 1, ..., n \). Mandatory and optional features below \( f \) remain as-is.

#### Directed Acyclic Graphs (\( T_{\text{DAG}} \))

Some feature modeling languages, such as FORM [23] and FEATUREB [18], use directed acyclic graphs opposed to trees. If a feature \( g \) has a decomposition type different from the defined ones (e.g., \( \lambda(g) = (a..b) \)), we set the decomposition type to an and-group (i.e., \( \lambda(g) = (a..n) \)) with \( n \) being the number of sub-features of \( g \) and add the following complex constraint:

\[
g \Rightarrow \bigwedge_{M \in P_{a,k}} \left( \bigwedge_{f \in M} f \wedge \bigwedge_{f' \in \{f' | g < f'\}} \neg f \right)
\]

with \( P_{a,k} = \{ A \in 2^{(f' \setminus g < f')} | a \leq |A| \leq b \} \) 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 (\( T_{\text{mutex}} \))

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 \( f_1, ..., f_n \), we keep the relationship \( f_1 < g \) and add an abstract feature \( aux_{i-1} \) for each \( f_2, ..., f_n \) such that \( f_1 < aux_{i-1} \). Finally, we add constraints \( g \equiv aux_{i-1} \) for all \( i = 2, ..., n \).

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 \( L_R \). 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.
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 simple requires constraint (i.e., \( f_1 \iff f_2 \)), 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., 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 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.
Is There a Mismatch between Real-World Feature Models and Product-Line Research?

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., EMIToolkit) 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.

---

299
Table 2: Overview of evaluated feature models including number of features and constraints before and after applying our constraint elimination approach.

<table>
<thead>
<tr>
<th>Feature Model</th>
<th>Features</th>
<th>Constraints</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automotive z.1</td>
<td>14,010</td>
<td>666</td>
<td>17.3%</td>
</tr>
<tr>
<td>Automotive z.2</td>
<td>17,742</td>
<td>914</td>
<td>16.2%</td>
</tr>
<tr>
<td>Automotive z.3</td>
<td>18,434</td>
<td>1,300</td>
<td>19.0%</td>
</tr>
<tr>
<td>Automotive z.4</td>
<td>18,636</td>
<td>1,369</td>
<td>19.4%</td>
</tr>
<tr>
<td>dTUI</td>
<td>96</td>
<td>14</td>
<td>19.0%</td>
</tr>
<tr>
<td>EmbToolkit 1.16</td>
<td>854</td>
<td>123</td>
<td>19.4%</td>
</tr>
<tr>
<td>BusyBox 1.18</td>
<td>1,179</td>
<td>323</td>
<td>17.3%</td>
</tr>
<tr>
<td>EmbToolkit 2</td>
<td>513</td>
<td>56</td>
<td>19.4%</td>
</tr>
<tr>
<td>uClinux-base</td>
<td>380</td>
<td>3455</td>
<td>17.3%</td>
</tr>
<tr>
<td>uClibc</td>
<td>1,580</td>
<td>197</td>
<td>17.3%</td>
</tr>
<tr>
<td>Linux 2.6.33.3</td>
<td>6,467</td>
<td>3,455</td>
<td>17.3%</td>
</tr>
<tr>
<td>BusyBox 1.22</td>
<td>816</td>
<td>816</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.23</td>
<td>1,178</td>
<td>1,178</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.22</td>
<td>3,455</td>
<td>3,455</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.23</td>
<td>816</td>
<td>816</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.18</td>
<td>513</td>
<td>513</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.16</td>
<td>854</td>
<td>854</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.15</td>
<td>1,179</td>
<td>1,179</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.14</td>
<td>1,580</td>
<td>1,580</td>
<td>0%</td>
</tr>
<tr>
<td>BusyBox 1.13</td>
<td>6,467</td>
<td>6,467</td>
<td>0%</td>
</tr>
<tr>
<td>All CDL Models</td>
<td>17,351</td>
<td>9,148</td>
<td>17.3%</td>
</tr>
</tbody>
</table>

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%

1 FeatureDE model, 2 RConfig, 3 Min ≤ Mean ≤ 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 BaSTV [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.
Is There a Mismatch between Real-World Feature Models and Product-Line Research?

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 and later versions. The Scala project is an extended version of the Linux Variability Analysis Project, 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. 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 .config 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


