Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems

Reimar Schröter,[1] Sebastian Krieter,[1] Thomas Thüm,[2] Fabian Benduhn,[1] and Gunter Saake[1]
[1] University of Magdeburg, Germany
[2] TU Braunschweig, Germany

ABSTRACT

Today's software systems are often customizable by means of load-time or compile-time configuration options. These options are typically not independent and their dependencies can be specified by means of feature models. As many industrial systems contain thousands of options, the maintenance and utilization of feature models is a challenge for all stakeholders. In the last two decades, numerous approaches have been presented to support stakeholders in analyzing feature models. Such analyses are commonly reduced to satisfiability problems, which suffer from the growing number of options. While first attempts have been made to decompose feature models into smaller parts, they still require to compose all parts for analysis. We propose the concept of a feature-model interface that only consists of a subset of features, typically selected by experts, and hides all other features and dependencies. Based on a formalization of feature-model interfaces, we prove compositionality properties. We evaluate feature-model interfaces using a three-month history of an industrial feature model from the automotive domain with 18,616 features. Our results indicate performance benefits especially under evolution as often only parts of the feature model need to be analyzed again.

CCS Concepts

•**Software and its engineering** → *Software product lines; Formal software verification; Feature interaction; Abstraction, modeling and modularity;*

Keywords

Configurable Software, Software Product Line, Variability Modeling, Feature Model, Modularity, Compositionality

1. INTRODUCTION

There is a growing need to customize software. This demand is often based on conflicting functional and non-functional requirements of each customer. Systematic reuse between customized software systems can be achieved by means of load-time or compile-time variability [7]. Common and variable artifacts of such configurable software are specified in terms of features, each representing a unit of functionality. However, not all features are compatible with one another, and some features force the existence of others. Feature models are typically used to describe all valid combinations of features [24], and can be represented as a hierarchical model or as a propositional formula [8].

As feature models specify valid combinations of features, they influence all phases of the development process for configurable software from requirements engineering to verification. In particular, all software analyses, such as type checking (e.g., [16, 45]) or model checking (e.g., [14, 28]), have to incorporate feature models to produce sound and complete results [46]. Hence, we are interested in all defects for valid feature combinations while defects for invalid combinations are not of interest. Besides such lifting of existing analyses, also numerous new analyses devoted to feature models have been presented [9]. For instance, it is intended that all features occur in at least one valid combination [24]. All these analyses have in common that they are reduced to one or more satisfiability problems by translating the feature model into a propositional formula [9, 46]. Hence, when the feature model evolves, usually the complete analysis has to be executed again even if the feature model changes only slightly.

This work is motivated by recent experiences in applying our feature-modeling tool FeatureIDE [47] to real feature models of our industrial partner. Dozens of stakeholders maintain a feature model from the automotive domain with 18,616 features and they face scalability problems for analyses, especially as they need to analyze the complete model again after every change. In addition, they expect that the feature model grows even further, which is also known from other industrial models [11], and from the Linux kernel with thousands of features [30]. We studied existing approaches to compose feature models from smaller parts [3, 6, 12, 13, 36, 41] but none of them considers compositional reasoning. That is, even if only a single part changes, we still have to compose all parts and perform all analyses again.

We propose to use feature-model composition with feature-model interfaces to enable compositional analyses. A feature-model interface is a feature model with a subset of all features that are selected and intended to be used by domain experts in a specific composition scenario. Similar to interfaces of programming languages, feature-model interfaces hide information (i.e., features) and can be used instead of the original feature model. As result, the feature-model in-

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884823

terfaces can be considered as placeholder and reduce the complexity of the composed model. In contrast to programming interfaces, the consumer defines the interface and not the provider. In this paper, we prove compositionality properties for feature-model interfaces. In addition, we illustrate their benefits by applying a time-consuming feature-model analysis to our industrial feature model. In particular, we make the following contributions:

- We formally define and illustrate feature-model composition and feature-model interfaces.
- We prove compositionality between the analysis results of compositions with and without feature-model interfaces.
- We evaluate the potential of feature-model interfaces for the three-month evolution of a real-world feature model.

2. FEATURE MODELS AND ANALYSES

In this section, we introduce feature models, formalize them, and present challenges regarding their correctness. Based on this, we also formalize common analyses that we investigate in the remainder of this paper.

2.1 Feature Models

A feature model describes a set of features and their valid combinations. Typically, feature diagrams are used to represent feature models graphically by arranging the features in a tree structure with additional cross-tree constraints to describe their dependencies [8, 24]. In Figure 1, we illustrate different types of feature dependencies by using the example of feature model *Index*, which represents a set of index structures to optimally support direct access of data items in a database. An index can only support one data type at a time. Thus, the features *Int*, *Double*, and *Float* are arranged in an alternative group. Furthermore, the developer can choose one or both of the search algorithms k-nearest neighbor (*Knn*) and *Range*. Since the query algorithms are independent of each other, they are represented by an or group. In addition, it is optional to force unique keys in an index structure for which we include an optional feature *UniqueKeys*. Since it is only possible to support unique keys for integer values, the model includes the additional cross-tree constraint $UniqueKeys \Rightarrow Int$ as propositional formula.

Besides feature diagrams, some other representations of feature models exist, such as propositional formulas, textual representations, or an enumeration of all valid configurations. To simplify our proofs, we formalize feature models as the set of all valid configurations:

Definition 1. *A feature model $\mathcal{M}_x$ is a tuple $(\mathcal{F}_x, \mathcal{P}_x)$, where (a) $\mathcal{F}_x$ is a set of features, and (b) $\mathcal{P}_x$ is a set of products with $\mathcal{P}_x \subseteq 2^{\mathcal{F}_x}$.*

In Figure 1, we exemplify this definition using feature model *Index*. However, this representation does not scale well in an actual implementation of any analysis and, thus, we rely on a conventional feature-model representation (i.e., a propositional formula) for our evaluation.

2.2 Feature-Model Analyses

In industry, feature models may consist of thousands of features [11]. Thus, automated consistency checks for large-scale feature models are important. Benavides et al. present an overview of automated analyses and consider them as an information extraction process that is executed in two steps [9]. First, an analysis tool translates the feature model into a specific representation (e.g., a propositional formula). Second, a specific algorithm uses a corresponding solver to perform the analysis. In the following, we define several analyses based on our feature-model formalization and illustrate their usage with feature model $\mathcal{M}_{Index}$.

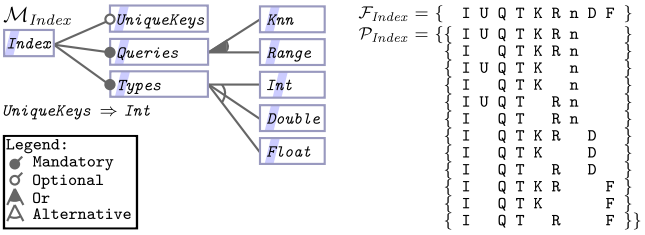

Figure 1: Feature model $\mathcal{M}_{Index} = (\mathcal{F}_{Index}, \mathcal{P}_{Index})$ (highlighted characters of the feature diagram are used to represent $\mathcal{F}_{Index}$ and $\mathcal{P}_{Index}$).

Definition 2. *Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathbb{M}$ the set of all feature models of the universe, then*

$$void = \{\mathcal{M}_x \in \mathbb{M} \mid \mathcal{P}_x = \emptyset\} \tag{2.1}$$

$$core(\mathcal{M}_x) = \bigcap_{p \in \mathcal{P}_x} p \tag{2.2}$$

$$dead(\mathcal{M}_x) = \mathcal{F}_x \setminus \bigcup_{p \in \mathcal{P}_x} p \tag{2.3}$$

$$pConf(\mathcal{M}_x) = \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists p \in \mathcal{P}_x : \mathcal{F}_S \subseteq p \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_x \setminus p\} \tag{2.4}$$

$$aSet(\mathcal{M}_x) = \{p \in aSub(\mathcal{M}_x) \mid \forall q \in aSub(\mathcal{M}_x) : p \not\subset q\} \tag{2.5}$$

$$aSub(\mathcal{M}_x) = \{q \mid q \neq \emptyset, \mathcal{P}_x \neq \emptyset, \forall p \in \mathcal{P}_x : q \subseteq p \vee q \subseteq \mathcal{F}_x \setminus p\} \tag{2.6}$$

A feature model is *void* if and only if it represents no products [8, 9, 24]. The analysis is particularly helpful for huge feature models with thousands of features to check the feature-model correctness [8, 9, 24, 50]. Using our formalization (see *void* in Eq. 2.1), we get the result that feature model $\mathcal{M}_{Index}$ is not void, because it is not contained in the set of all void feature models (i.e., $\mathcal{M}_{Index} \notin void$).

A *core feature* is a feature that is included in all products of a non-void product line [9, 52]. The analysis can be used to determine feature priorities, as it may be useful to develop core features first [9, 51]. Using our formalization (see $core(\mathcal{M}_x)$ in Eq. 2.2) with feature model $\mathcal{M}_{Index}$, we obtain the set $\{Index, Queries, Types\}$.

A feature of a feature model is a *dead feature* if and only if it is not part of any valid product of a non-void product line [9, 24]. The analysis is particularly useful to identify contradictions in feature models [21]. Furthermore, it avoids implementing features that are not part of any product. For feature model $\mathcal{M}_{Index}$, the application of function *dead* (see $dead(\mathcal{M}_x)$ in Eq. 2.3) results in an empty set and, thus, the feature model does not contain any dead features.

A *partial configuration* is a tuple consisting of a set of selected features $\mathcal{F}_S$ and a set of deselected features $\mathcal{F}_D$ with the restriction $\mathcal{F}_S \cup \mathcal{F}_D \subseteq \mathcal{F}_x$ and $\mathcal{F}_S \cap \mathcal{F}_D = \emptyset$. If the union of $\mathcal{F}_S$ and $\mathcal{F}_D$ is equal to $\mathcal{F}_x$ the tuple represents a full configuration, which describes exactly one product [9]. Hence, a full configuration is included in our definition as a special case. The analysis of partial configurations investigates

whether a partial configuration fulfills all the dependencies of the corresponding feature model [8, 9, 24]. Using our formalization (see $pConf(\mathcal{M}_x)$ in Eq. 2.4) for feature model $\mathcal{M}_{Index}$, the configuration ($\{$`I,Q,T,K,n`$\}$, $\{$`U,R,D,F`$\}$) is part of the resulting configurations of function *pConf* and, thus, a valid configuration. By contrast, the partial configuration ($\{$`I,U,D`$\}, \emptyset$) is invalid because it does not conform to the cross-tree constraint and alternative group.

An *atomic set* is a non-empty set of features, which is either completely included or completely absent in each product. The analysis is used to reduce the size of a feature model as input for further analyses [9, 42, 53]. However, most implementations only consider a mandatory feature and its parent as atomic set [9, 53]. By contrast, similar to Durán et al. [18], we formally define the function *aSet (atomic Set)*, which uses a feature model $\mathcal{M}_x$ as input and returns the set of all atomic sets (see $aSet(\mathcal{M}_x)$ in Eq. 2.5). To ease the definition of atomic sets with function *aSet*, we use a second function *aSub (atomic Subset)* that determines all *atomic subsets*, i.e., all subsets of features that are either completely included or completely absent in each product. However, function *aSet* is used to find all super sets in *aSub*'s result, i.e., the function removes all subsets that are completely contained in other sets. Hence, the set of atomic sets is always a subset of the set of atomic subsets (i.e., $aSet(\mathcal{M}_x) \subseteq aSub(\mathcal{M}_x)$). As an example, we use function *aSet* to determine all atomic sets of feature model $\mathcal{M}_{Index}$. Besides the atomic sets with one feature, there is one atomic set containing all core features.

3. PROBLEM STATEMENT

As seen in the Linux kernel, in the application scenario of our industrial partner, and in other case studies [11, 44, 47], the complexity of feature models can be challenging for humans and machines. In the following, we explain some of the most important problems.

Manual construction and maintenance of a feature model with thousands of features is almost impossible because the size is overwhelming and blurs important feature dependencies. Furthermore, feature diagrams do not scale for this feature-model size. Therefore, decomposition is used to handle large feature models [11, 35]. Using this strategy, different groups of domain experts can work on smaller feature models, which are easier to understand and to visualize. If we use feature-model composition to reuse existing feature models in another one, it is possible that only some features are needed to describe or understand the feature-model dependencies. A complete feature-model reuse can affect the comprehension of the composition and again blur the important dependencies.

Decomposition of a large feature model into smaller fragments is one state-of-the-art strategy to ease human comprehensibility and maintainability. However, to use the previously described analyses for a large, decomposed feature model, it is necessary to combine its fragments into a representation that is analyzable using existing techniques [41]. The result is again a large feature model for which not all analyses scale. Thus, the scalability problem of complex analyses in the context of large feature models is not solved yet. A compositional procedure in which we can reuse the results of analyses in smaller feature models for the computation in a feature-model composition is desirable.

The previous problems are by themselves hard to solve but if we take evolution of decomposed feature models into account, the situation gets even worse. In this scenario, we consider changes in feature-model fragments, where we need to re-execute all desired analyses. In particular, it is challenging to reuse existing analysis results of the previous feature-model version and, thus, it is necessary to re-compute the complete analysis. However, evolutionary changes are not unusual [2, 11, 17], e.g., within three months, the feature model of our industrial partner was extended by more than 4,000 features. Therefore, it is desirable to reduce the amount of (complete) re-computations, when feature-model fragments change.

4. COMPOSITIONALITY BASICS

As we aim to reduce the mentioned problems, we propose to use a combination of feature-model composition and feature-model interfaces. Based on their formalization, we prove compositionality properties in the remaining paper.

4.1 Feature-Model Composition

Multiple mechanisms exist that allow us to combine feature models [3, 6, 12, 13, 36]. In this paper, we consider the composition through aggregation, i.e., by inclusion of one feature model as an instance in another feature model [36].

In our running example, we want to create an instance of feature model $\mathcal{M}_{Index}$ by connecting it to the feature *Access* in the database feature model $\mathcal{M}_{DBMS}$. To further specify their relationship, we add two additional cross-tree constraints to the resulting feature model. We depict the composition result on the left side of Figure 2 (feature model $\mathcal{M}_C$ is used to describe the dependencies of the models).

We define semantics of the composition as follows:

Definition 3. *Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, and $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$ be feature models with $\mathcal{F}_C \subseteq \mathcal{F}_x \cup \mathcal{F}_y$. We define the function* composition $\circ$ *using $\mathcal{M}_x, \mathcal{M}_y$, and $\mathcal{M}_C$ with infix notation $\circ_{\mathcal{M}_C}$ based on the join function $\bullet$ and function $\mathcal{R}$ to achieve the composed feature model $\mathcal{M}_{x/y}$:*

$$\mathcal{M}_{x/y} = \circ(\mathcal{M}_x, \mathcal{M}_y, \mathcal{M}_C) = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y = (\mathcal{M}_x \bullet \mathcal{R}(\mathcal{M}_y)) \bullet \mathcal{M}_C \quad (3.1)$$

$$\mathcal{R}(\mathcal{M}_y) = \mathcal{R}((\mathcal{F}_y, \mathcal{P}_y)) = (\mathcal{F}_y, \mathcal{P}_y \cup \{\emptyset\}) \quad (3.2)$$

$$\mathcal{M}_x \bullet \mathcal{M}_y = (\mathcal{F}_x, \mathcal{P}_x) \bullet (\mathcal{F}_y, \mathcal{P}_y) = (\mathcal{F}_x \cup \mathcal{F}_y, \{p \cup q \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \quad (3.3)$$

The definition of the composition function $\circ$ is based on the functions $\bullet$ (*join*) and R (*remove core property*). Function R takes one feature model as input and converts it to a new feature model in which the empty product is a valid product. Thus, the feature set is identical to the input feature model and the set of products is extended by the empty set. R is used in function $\circ$ to ensure that $\mathcal{M}_y$'s core features are not necessarily core in the composed feature model.

The join function $\bullet$ takes two feature models as input and returns a new combined feature model, which is a merge of all input information (i.e., features and products). In detail, the resulting feature model consists of all features from the input feature models. To combine the product sets of both input models, we use an operation that is similar to a join as known from relational algebra [15]. Like the join, we only combine two products if the additional condition

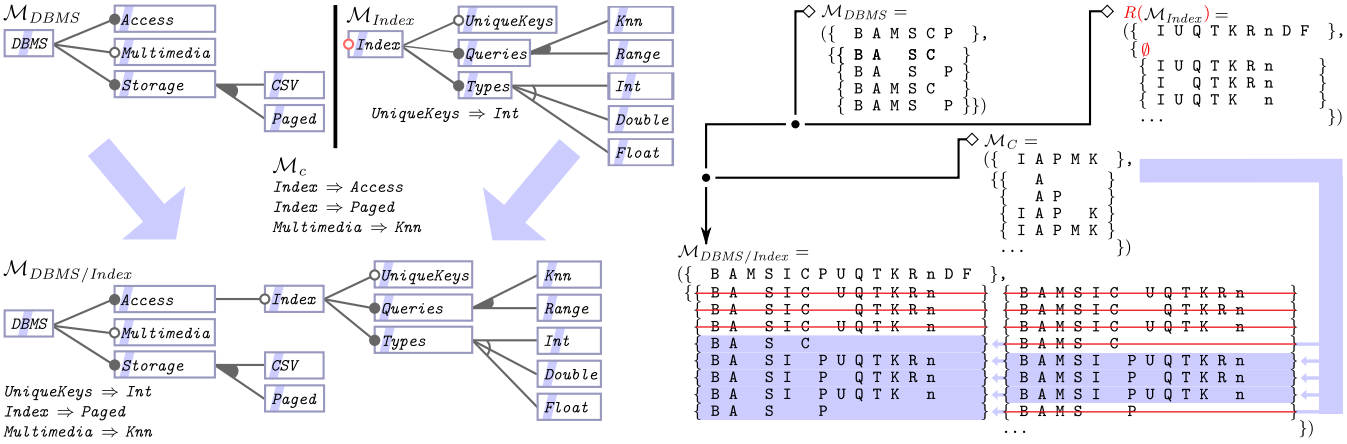

Figure 2: Composition of the feature models $\mathcal{M}_{DBMS}$ and $\mathcal{M}_{Index}$ using the feature model $\mathcal{M}_C$, which describes dependencies between both feature models.

$p \cap \mathcal{F}_y = q \cap \mathcal{F}_x$ is fulfilled. Consequently, the function's result is a feature model that conforms to our Definition 1. In particular, if both feature sets are disjoint, the condition $p \cap \mathcal{F}_y = q \cap \mathcal{F}_x = \emptyset$ is always true. Hence, our function behaves similar to a cross product from relational algebra and creates all combinations between both product sets.

The composition function $\circ$, based on the functions R and $\bullet$, uses three feature models as input to create a new one. The second feature model $\mathcal{M}_y$ is instantiated in the first feature model $\mathcal{M}_x$. The third feature model $\mathcal{M}_C$ describes a parent-child relationship and other inter-model constraints between $\mathcal{M}_x$ and $\mathcal{M}_y$ in order to connect both models. While function $\circ$ allows us to combine arbitrary feature models, our proofs (cf. Section 5) are based on the assumption that $\mathcal{F}_x$ and $\mathcal{F}_y$ do not share features (i.e., $\mathcal{F}_x \cap \mathcal{F}_y = \emptyset$).

In Figure 2, we exemplify all three functions using $\mathcal{M}_{DBMS}$ and $\mathcal{M}_{Index}$. To instantiate feature model $\mathcal{M}_{Index}$ in feature model $\mathcal{M}_{DBMS}$, we have to transform feature model $\mathcal{M}_{Index}$ using our function R and create all product combinations using our join function (i.e., $\mathcal{M}_{DBMS} \bullet \mathcal{R}(\mathcal{M}_{Index})$). This results in additional product combinations that are not part of the final feature model $\mathcal{M}_{DBMS/Index}$ due to the absence of feature model $\mathcal{M}_C$. We depict the intermediate result in Figure 2. Finally, we eliminate all unintended products using the join function $\bullet$ with the intermediate result and the feature model $\mathcal{M}_C$, which contains the desired parent-child dependency and the two cross-tree constraints. The highlighted products of feature model $\mathcal{M}_{DBMS/Index}$ represent the final result of our composition function $\circ$.

4.2 Feature-Model Interfaces

We now formally define feature-model interfaces and prove algebraic properties of feature-model interfaces that we need for our compositionality proofs.

4.2.1 Formalization of Feature-Model Interfaces

We define a feature-model interface as follows:

Definition 4. *A feature model $\mathcal{M}_{Int} = (\mathcal{F}_{Int}, \mathcal{P}_{Int})$ is an interface of feature model $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ denoted as $\mathcal{M}_{Int} \preceq \mathcal{M}_x$, if and only if* (a) $\mathcal{F}_{Int} \subseteq \mathcal{F}_x$ *and* (b) $\mathcal{P}_{Int} = \{p \cap \mathcal{F}_{Int} \mid p \in \mathcal{P}_x\}$.

If two feature models $\mathcal{M}_x$ and $\mathcal{M}_{Int}$ do not fulfill this definition (i.e., $\mathcal{M}_{Int} \not\preceq \mathcal{M}_x$), we call $\mathcal{M}_{Int}$ incompatible to $\mathcal{M}_x$.

From Definition 4, we can infer that for each pair $\mathcal{M}_x$ and $\mathcal{F}_{Int}$ there exists exactly one feature-model interface. A vital characteristic of a feature-model interface is that it is a feature model itself. Therefore, we are able to use an interface instead of a feature model for composition. In detail, the feature-model interface $\mathcal{M}_{Int}$ has a possibly reduced set of features compared to feature model $\mathcal{M}_x$. Furthermore, each product of $\mathcal{M}_{Int}$ is a subset of a product of $\mathcal{M}_x$, including only features from $\mathcal{F}_{Int}$. Moreover, each product of $\mathcal{M}_x$ is a super set of one or more products of $\mathcal{M}_{Int}$.

Corollary 5. $\forall p \in \mathcal{P}_{Int} \exists q \in \mathcal{P}_x : p = q \cap \mathcal{F}_{Int}$
$\forall q \in \mathcal{P}_x \exists p \in \mathcal{P}_{Int} : p = q \cap \mathcal{F}_{Int}$

For our theoretical investigation of feature-model interfaces, we define a function S (*slice*) (similar to the slice operator proposed by Acher et al. [4]) that allows us to generate a feature-model interface by removing a given set of features, which are of no interest for a specific target domain.

Definition 6. *We define a function S that takes a feature model $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ and a set of features $\mathcal{F}_R$ as input and returns a feature model $\mathcal{M}_{Int}$ with $\mathcal{M}_{Int} \preceq \mathcal{M}_x$.*

$$\mathcal{M}_{Int} = S(\mathcal{M}_x, \mathcal{F}_R) = (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\})$$

For our running example, we want to reuse $\mathcal{M}_{Index}$ as enhancement of $\mathcal{M}_{DBMS}$. However, some features are not of our interest and, thus, we apply function S on $\mathcal{M}_{Index}$ with the set of features $\mathcal{F}_R$ to be removed, $\mathcal{F}_R = \{$*Range*, *UniqueKeys*, *Float*$\}$ (cf. Figure 3). In practice, $\mathcal{F}_R$ depends on the specific reuse scenario in agreement with the stakeholders.

4.2.2 Algebraic Properties of Interfaces

Next, we take a look at certain properties of function S, which we will use in our proofs for compositionality. In detail, we investigate its right identity for certain feature sets and the distributivity with the functions $\bullet$ and R.

Right Identity. A feature set $\mathcal{F}_R$ is a right identity element to S if $\mathcal{F}_x$ does not contain any feature from $\mathcal{F}_R$. As result, the application of S has no effect on a feature model that does not contain a feature of the feature set $\mathcal{F}_R$.

Lemma 7. *Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{F}_R$ a set of features with $\mathcal{F}_x \cap \mathcal{F}_R = \emptyset$, then $S(\mathcal{M}_x, \mathcal{F}_R) = \mathcal{M}_x$.*

Proof. *As the intersection of $\mathcal{F}_x$ and $\mathcal{F}_R$ is the empty set, there will be no feature that is removed from the set $\mathcal{F}_x$. The result is the identical feature set $\mathcal{F}_x$. Similarly, the intersection between each product and the set of features $\mathcal{F}_R$ is also empty and, thus, each product will be the same as before.*

$$S((\mathcal{F}_x, \mathcal{P}_x), \mathcal{F}_R) = ((\mathcal{F}_x \setminus \mathcal{F}_R), \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\}) \quad (7.1)$$
$$= (\mathcal{F}_x, \mathcal{P}_x) = \mathcal{M}_x \quad (7.2)$$

□

Distributivity of $\bullet$ and S. The order in which we apply the functions $\bullet$ and S is not relevant for the result.

Lemma 8. *Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$ be feature models and $\mathcal{F}_R$ a set of features, then*

$$S(\mathcal{M}_x \bullet \mathcal{M}_y, \mathcal{F}_R) = S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(\mathcal{M}_y, \mathcal{F}_R).$$

Proof. *In general, we separate the application of the function S on each part of the composed feature model so that we can apply function $\bullet$ later on.*

$$S((\mathcal{F}_x, \mathcal{P}_x) \bullet (\mathcal{F}_y, \mathcal{P}_y), \mathcal{F}_R) \quad (8.1)$$
$$= (\mathcal{F}_z, \mathcal{P}_z) \quad (8.2)$$
$$= ((\mathcal{F}_x \cup \mathcal{F}_y) \setminus \mathcal{F}_R, \mathcal{P}_z) \quad (8.3)$$
$$= ((\mathcal{F}_x \setminus \mathcal{F}_R) \cup (\mathcal{F}_y \setminus \mathcal{F}_R), \mathcal{P}_z) \quad (8.4)$$

Next, without loss of generality, we introduce the sets r and s to represent the results of function S, which are then used as input for function $\bullet$.

$$= (\mathcal{F}_z, \{(p \cup q) \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \quad (8.5)$$
$$= (\mathcal{F}_z, \{(p \setminus \mathcal{F}_R) \cup (q \setminus \mathcal{F}_R) \mid p \in \mathcal{P}_x, q \in \mathcal{P}_y, p \cap \mathcal{F}_y = q \cap \mathcal{F}_x\}) \quad (8.6)$$
$$\overset{(Definition\ 4)}{=} (\mathcal{F}_z, \{r \cup s \mid r \in \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\}, s \in \{q \setminus \mathcal{F}_R \mid q \in \mathcal{P}_y\}, r \cap \mathcal{F}_y = s \cap \mathcal{F}_x\}) \quad (8.7)$$
$$\overset{(Definition\ 3)}{=} S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(\mathcal{M}_y, \mathcal{F}_R)$$

□

Distributivity of R and S. Finally, we prove that the order in which we apply the functions R and S is not relevant.

Lemma 9. *Let $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$ be a feature model and $\mathcal{F}_R$ a set of features, then $S(R(\mathcal{M}_x), \mathcal{F}_R) = R(S(\mathcal{M}_x, \mathcal{F}_R))$.*

Proof. *Function R adds the empty set to the set of products. To prove the interaction of R with S, it is necessary to extract this empty set from the input feature model of S.*

$$S(R(\mathcal{M}_x), \mathcal{F}_R) = (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in (\mathcal{P}_x \cup \{\emptyset\})\}) \quad (9.1)$$
$$= (\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\} \cup \{\emptyset\}) \quad (9.2)$$
$$= R((\mathcal{F}_x \setminus \mathcal{F}_R, \{p \setminus \mathcal{F}_R \mid p \in \mathcal{P}_x\})) \quad (9.3)$$
$$= R(S(\mathcal{M}_x, \mathcal{F}_R))$$

□

5. COMPOSITIONALITY IN THEORY

First, we present the general idea of compositionality that is based on feature-model composition and feature-model interfaces. Second, we show the potential of this combination using the presented feature-model analyses of Section 2.

5.1 Compositionality Principle

To introduce our general concept of feature-model compositionality, we assume that two feature models $\mathcal{M}_x$ and $\mathcal{M}_y$ are composed to $\mathcal{M}_{x/y} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y$. Typically, not all features of feature model $\mathcal{M}_y$ are of interest for the composition with feature model $\mathcal{M}_x$. Given the knowledge about those features, it is possible to create a feature-model interface $\mathcal{M}_{Int}$ based on $\mathcal{M}_y$ (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_y$) with all features of interest. Now, it is possible to use feature model $\mathcal{M}_{Int}$ instead of feature model $\mathcal{M}_y$ for feature-model composition with $\mathcal{M}_x$ (i.e., $\mathcal{M}_{x/Int} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int}$) and subsequently use $\mathcal{M}_{x/Int}$ for automated analyses. However, analysis results for $\mathcal{M}_{x/Int}$ might differ from the results for $\mathcal{M}_{x/y}$, due to the reduced feature set of $\mathcal{M}_{Int}$. Therefore, we identify and prove specific relations between analysis results based on $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$ for each analysis of Section 2. For instance, the analysis result of dead features for $\mathcal{M}_{x/Int}$ is a subset of the result for $\mathcal{M}_{x/y}$. However, the main profit of this dependency exists in an evolution scenario. If a new version of $\mathcal{M}_y$ exists that still conforms to the interface, the results for $\mathcal{M}_{x/Int}$ are identical.

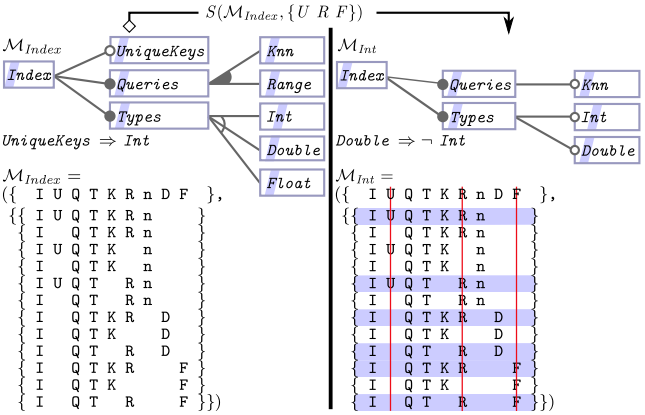

Figure 3: Application of function S on feature model $\mathcal{M}_{Index}$. The highlighted products are part of the resulting feature-model interface $\mathcal{M}_{Int}$.

For our proofs regarding the analysis-result relations of feature model $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$, we have to consider an additional property. In detail, we prove that a feature-model composition based on a feature-model interface $\mathcal{M}_{Int}$ is itself a feature-model interface in relation to a composition based on $\mathcal{M}_y$ (i.e., $\mathcal{M}_{x/Int} \preceq \mathcal{M}_{x/y}$). This property is based on the assumption, that we only remove features from the set $\mathcal{F}_y$ whereas the feature sets $\mathcal{F}_x$ and $\mathcal{F}_C$ remain unchanged.

Lemma 10. *Let $\mathcal{M}_{x/y} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y$, $\mathcal{M}_{x/Int} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int}$ be composed feature models based on the models $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$, $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R)$ with $\mathcal{F}_R \cap \mathcal{F}_x = \mathcal{F}_R \cap \mathcal{F}_C = \emptyset$, then: $\mathcal{M}_{x/Int} \preceq \mathcal{M}_{x/y}$.*

Proof. *Given the algebraic properties of the function S and the definition of our composition function $\circ_{\mathcal{M}_C}$, the following relations hold:*

$$\mathcal{M}_{x/y} \succeq S(\mathcal{M}_{x/y}, \mathcal{F}_R) \quad (10.1)$$
$$= S(\mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y, \mathcal{F}_R) \quad (10.2)$$
$$\overset{(Eq.\ 3.1)}{=} S((\mathcal{M}_x \bullet R(\mathcal{M}_y)) \bullet \mathcal{M}_C, \mathcal{F}_R) \quad (10.3)$$
$$\overset{(Lemma\ 8)}{=} (S(\mathcal{M}_x, \mathcal{F}_R) \bullet S(R(\mathcal{M}_y), \mathcal{F}_R)) \bullet S(\mathcal{M}_C, \mathcal{F}_R) \quad (10.4)$$
$$\overset{(Lemma\ 7)}{=} (\mathcal{M}_x \bullet S(R(\mathcal{M}_y), \mathcal{F}_R)) \bullet \mathcal{M}_C \quad (10.5)$$

$$\overset{(Lemma\ 9)}{=} (\mathcal{M}_x \bullet R(S(\mathcal{M}_y, \mathcal{F}_R))) \bullet \mathcal{M}_C \quad (10.6)$$

$$\overset{(Definition\ 6)}{=} (\mathcal{M}_x \bullet R(\mathcal{M}_{Int}))) \bullet \mathcal{M}_C \quad (10.7)$$

$$\overset{(Eq.\ 3.1)}{=} \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int} \quad (10.8)$$

$$= \mathcal{M}_{x/Int} \qquad \square$$

As a result of this proof, we can consider feature model $\mathcal{M}_{x/Int}$ as an ordinary feature-model interface of $\mathcal{M}_{x/y}$, in which the knowledge about the initial composition is not relevant. Thus, it is sufficient to prove the analysis-result relations between feature model $\mathcal{M}_{Int}$ and $\mathcal{M}_y$ because the same dependency holds for feature model $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$.

5.2 Compositionality of Existing Analyses

In this section, we investigate the compositionality of the analyses of *void feature model*, *core features*, *dead features*, *valid partial configurations*, and *atomic sets*. For each analysis, we first examine the analysis-result relation between feature model $\mathcal{M}_y$ and $\mathcal{M}_{Int}$ followed by an investigation of the composed feature models using the following two premises:

Premise 1. *Let $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$ be a feature model and $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R) = (\mathcal{F}_{Int}, \mathcal{P}_{Int})$ its feature-model interface (i.e., $\mathcal{M}_{Int} \preceq \mathcal{M}_y$).*

Premise 2. *Let $\mathcal{M}_{x/y} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_y$, $\mathcal{M}_{x/Int} = \mathcal{M}_x \circ_{\mathcal{M}_C} \mathcal{M}_{Int}$ be composed feature models based on the feature models $\mathcal{M}_x = (\mathcal{F}_x, \mathcal{P}_x)$, $\mathcal{M}_y = (\mathcal{F}_y, \mathcal{P}_y)$, $\mathcal{M}_C = (\mathcal{F}_C, \mathcal{P}_C)$, $\mathcal{M}_{Int} = S(\mathcal{M}_y, \mathcal{F}_R)$ with $\mathcal{F}_R \cap \mathcal{F}_x = \mathcal{F}_R \cap \mathcal{F}_C = \emptyset$.*

5.2.1 Void Feature Model

With respect to Premise 1, $\mathcal{M}_{Int}$ is void if and only if $\mathcal{M}_y$ is void.

Theorem 11. $\mathcal{M}_y \in void \Leftrightarrow \mathcal{M}_{Int} \in void$.

Proof. *With Corollary 5, the following equivalences hold:*

$$\mathcal{M}_y \in void \Leftrightarrow \mathcal{P}_y = \emptyset \quad (11.1)$$

$$\overset{(Corollary\ 5)}{\Leftrightarrow} \mathcal{P}_{Int} = \emptyset \quad (11.2)$$

$$\Leftrightarrow \mathcal{M}_{Int} \in void \qquad \square$$

Based on this knowledge and Premise 2, we deduce that a feature model $\mathcal{M}_{x/Int}$ is void if and only if $\mathcal{M}_{x/y}$ is void.

Theorem 12. $\mathcal{M}_{x/y} \in void \Leftrightarrow \mathcal{M}_{x/Int} \in void$.

Proof. *From Lemma 10 and Theorem 11, we infer that the same analysis-result relation is also valid for $\mathcal{M}_{x/Int}$ and $\mathcal{M}_{x/y}$.* $\square$

5.2.2 Core Features

With respect to Premise 1, a feature $f \in \mathcal{F}_{Int}$ is a core feature of $\mathcal{M}_{Int}$ if and only if f is a core feature of $\mathcal{M}_y$.

Theorem 13. $core(\mathcal{M}_y) \cap \mathcal{F}_{Int} = core(\mathcal{M}_{Int})$.

Proof. *With Definition 4, the following equation holds:*

$$core(\mathcal{M}_{Int}) = \bigcap_{p \in \mathcal{P}_{Int}} p \quad (13.1)$$

$$\overset{(Definition\ 4)}{=} \bigcap_{p' \in \mathcal{P}_y} (p' \cap \mathcal{F}_{Int}) \quad (13.2)$$

$$= (\bigcap_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int} \quad (13.3)$$

$$= core(\mathcal{M}_y) \cap \mathcal{F}_{Int} \qquad \square$$

Therefore, we can conclude that, if a feature f is a core feature of $\mathcal{M}_{Int}$, it is also a core feature of $\mathcal{M}_y$. In addition, if we determine core features of $\mathcal{M}_y$ that are also part of $\mathcal{M}_{Int}$, these are also core features of feature model $\mathcal{M}_{Int}$.

$$f \in core(\mathcal{M}_{Int}) \Rightarrow f \in core(\mathcal{M}_y)$$

$$f \in core(\mathcal{M}_y) \cap \mathcal{F}_{Int} \Rightarrow f \in core(\mathcal{M}_{Int})$$

Using Theorem 13 and Premise 2, for composed feature models, we can deduce that a feature $f \in \mathcal{F}_{x/Int}$ is a core feature of $\mathcal{M}_{x/Int}$ if and only if f is a core feature in $\mathcal{M}_{x/y}$.

Theorem 14. $core(\mathcal{M}_{x/y}) \cap \mathcal{F}_{x/Int} = core(\mathcal{M}_{x/Int})$.

Proof. *Analogous to Theorem 12.* $\square$

5.2.3 Dead Features

In compliance with Premise 1, a feature $f \in \mathcal{F}_{Int}$ is a dead feature of $\mathcal{M}_{Int}$ if and only if f is a dead feature of $\mathcal{M}_y$.

Theorem 15. $dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} = dead(\mathcal{M}_{Int})$

Proof. *Based on Definition 4, the following equations hold:*

$$dead(\mathcal{M}_{Int}) = \mathcal{F}_{Int} \setminus \bigcup_{p \in \mathcal{P}_{Int}} p \quad (15.1)$$

$$\overset{(Definition\ 4)}{=} (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus (\bigcup_{p' \in \mathcal{P}_y} (p' \cap \mathcal{F}_{Int})) \quad (15.2)$$

$$= (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus ((\bigcup_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int}) \quad (15.3)$$

$$= (\mathcal{F}_y \setminus \bigcup_{p' \in \mathcal{P}_y} p') \cap \mathcal{F}_{Int} \quad (15.4)$$

$$= dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} \qquad \square$$

Therefore, if a feature f is a dead feature in $\mathcal{M}_{Int}$, it is also a dead feature in $\mathcal{M}_y$. Furthermore, if a feature f is a dead feature in feature model $\mathcal{M}_y$ and f is also part of $\mathcal{M}_{Int}$, it is also a dead feature in feature-model interface $\mathcal{M}_{Int}$.

$$f \in dead(\mathcal{M}_{Int}) \Rightarrow f \in dead(\mathcal{M}_y)$$

$$f \in dead(\mathcal{M}_y) \cap \mathcal{F}_{Int} \Rightarrow f \in dead(\mathcal{M}_{Int})$$

Again, we take a look into the relations of analysis results regarding feature-model compositions. Using Premise 2, a feature $f \in \mathcal{F}_{x/Int}$ is a dead feature of feature model $\mathcal{M}_{x/Int}$ if and only if f is a dead feature of $\mathcal{M}_{x/y}$.

Theorem 16. $dead(\mathcal{M}_{x/y}) \cap \mathcal{F}_{x/Int} = dead(\mathcal{M}_{x/Int})$

Proof. *Analogous to Theorem 12.* $\square$

5.2.4 Valid Partial Configurations

Regarding Premise 1, a configuration $C = (\mathcal{F}_S, \mathcal{F}_D)$ with $\mathcal{F}_S \subseteq \mathcal{F}_{Int}$, $\mathcal{F}_D \subseteq \mathcal{F}_{Int}$ is a valid partial configuration of $\mathcal{M}_{Int}$ if and only if C is a valid partial configuration of $\mathcal{M}_y$.

Theorem 17. $pConf(\mathcal{M}_{Int}) = \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)\}$

Proof. *With Definition 4, the following equation holds:*

$pConf(\mathcal{M}_{Int})$

$$\overset{(Definition\ 2)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists p \in \mathcal{P}_{Int} : \mathcal{F}_S \subseteq p \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_{Int} \setminus p\} \quad (17.1)$$

$$\overset{(Corollary\ 5)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \wedge \mathcal{F}_D \subseteq \mathcal{F}_{Int} \setminus (q \cap \mathcal{F}_{Int})\} \quad (17.2)$$

$$\overset{(\mathcal{F}_{Int} \subseteq \mathcal{F}_y)}{=} \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \wedge \mathcal{F}_D \subseteq (\mathcal{F}_y \cap \mathcal{F}_{Int}) \setminus (q \cap \mathcal{F}_{Int})\} \quad (17.3)$$

$$= \{(\mathcal{F}_S, \mathcal{F}_D) \mid \exists q \in \mathcal{P}_y : \mathcal{F}_S \subseteq q \cap \mathcal{F}_{Int} \wedge \mathcal{F}_D \subseteq (\mathcal{F}_y \setminus q) \cap \mathcal{F}_{Int}\} \quad (17.4)$$

$$= \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid \exists q \in \mathcal{P}_y : \mathcal{F}_S \subseteq q \ \wedge \mathcal{F}_D \subseteq \mathcal{F}_y \setminus q\} \quad (17.5)$$

$$\overset{(Definition\ 2)}{=} \{(\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)\}$$ □

As result, we know that each valid partial configuration of $\mathcal{M}_{Int}$ is also a valid partial configuration of $\mathcal{M}_y$ and valid partial configurations of $\mathcal{M}_y$ are also valid partial configurations of $\mathcal{M}_{Int}$ if $\mathcal{F}_S$ and $\mathcal{F}_D$ are intersected with $\mathcal{F}_{Int}$.

$$(\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_{Int}) \Rightarrow (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y)$$
$$(\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_y) \Rightarrow (\mathcal{F}_S \cap \mathcal{F}_{Int}, \mathcal{F}_D \cap \mathcal{F}_{Int}) \in pConf(\mathcal{M}_{Int})$$

Based on Theorem 17 and Premise 2, we consider the relationship of analysis results of composed feature models. Hence, a partial configuration with $\mathcal{F}_S \subseteq \mathcal{F}_{x/Int}$ and $\mathcal{F}_D \subseteq \mathcal{F}_{x/Int}$ is a valid partial configuration of $\mathcal{M}_{x/Int}$ if and only if $(\mathcal{F}_S, \mathcal{F}_D)$ is a valid partial configuration of $\mathcal{M}_{x/y}$.

Theorem 18. $pConf(\mathcal{M}_{x/Int}) = \{(\mathcal{F}_S \cap \mathcal{F}_{x/Int}, \mathcal{F}_D \cap \mathcal{F}_{x/Int}) \mid (\mathcal{F}_S, \mathcal{F}_D) \in pConf(\mathcal{M}_{x/y})\}$

Proof. *Analogous to Theorem 12.* □

5.2.5 Atomic Sets

With respect to Premise 1, a feature set A with $A \subseteq \mathcal{F}_y$ is an atomic subset with $A \cap \mathcal{F}_{Int}$ of feature model $\mathcal{M}_{Int}$ if and only if A is an atomic subset of feature model $\mathcal{M}_y$. We prove this relation using function *aSub*.

Theorem 19. $aSub(\mathcal{M}_{Int}) = \{q \cap \mathcal{F}_{Int} \mid q \in aSub(\mathcal{M}_y), q \cap \mathcal{F}_{Int} \neq \emptyset\}$

Proof. $aSub(\mathcal{M}_{Int})$

$$\overset{(Definition\ 2)}{=} \{q \mid q \neq \emptyset, \mathcal{P}_{Int} \neq \emptyset, \forall p \in \mathcal{P}_{Int} : (q \subseteq p) \vee (q \subseteq \mathcal{F}_{Int} \setminus p)\} \quad (19.1)$$

$$\overset{(Corollary\ 5)}{=} \{q \mid \mathcal{P}_y \neq \emptyset, \ q \neq \emptyset, \forall p \in \mathcal{P}_y : (q \subseteq p \cap \mathcal{F}_{Int}) \vee (q \subseteq (\mathcal{F}_y \setminus p) \cap \mathcal{F}_{Int})\} \quad (19.2)$$

$$= \{q \cap \mathcal{F}_{Int} \mid \mathcal{P}_y \neq \emptyset, q \neq \emptyset, q \cap \mathcal{F}_{Int} \neq \emptyset, \forall p \in \mathcal{P}_y : (q \subseteq p) \vee (q \subseteq \mathcal{F}_y \setminus p)\} \quad (19.3)$$

$$\overset{(Definition\ 2)}{=} \{q \cap \mathcal{F}_{Int} \mid q \in aSub(\mathcal{M}_y), q \cap \mathcal{F}_{Int} \neq \emptyset\}$$ □

Thus, we know that each atomic subset A of $\mathcal{M}_{Int}$ is also an atomic subset of $\mathcal{M}_y$. In addition, an atomic subset A of $\mathcal{M}_y$ intersected with $\mathcal{F}_{Int}$ is also an atomic subset of $\mathcal{M}_{Int}$.

$$A \in aSub(\mathcal{M}_{Int}) \Rightarrow A \in aSub(\mathcal{M}_y)$$
$$A \in aSub(\mathcal{M}_y) \Rightarrow (A \cap \mathcal{F}_{Int}) \in aSub(\mathcal{M}_{Int})$$

Using Theorem 19 and the Premise 2, we investigate the relation of atomic sets in composed feature models. In detail, a set A with $A \subseteq \mathcal{F}_{x/Int}$ is an atomic subset of $\mathcal{M}_{x/Int}$ if and only if A is an atomic subset of $\mathcal{M}_{x/y}$.

Theorem 20. $aSub(\mathcal{M}_{x/Int}) = \{q \cap \mathcal{F}_{x/Int} \mid q \in aSub(\mathcal{M}_{x/y}), q \cap \mathcal{F}_{x/Int} \neq \emptyset\}$

Proof. *Analogous to Theorem 12.* □

5.3 Discussion

To exemplify the obtained properties of compositionality with feature-model interfaces, we reconsider the identified problems of Section 3 and use our running example with the analysis of atomic sets as illustration. Of course, the approach is also applicable for other analyses.

First, we considered the problem of feature-model scalability for humans. Compared to the composed feature model $\mathcal{M}_{DBMS/Index}$, $\mathcal{M}_{DBMS/Int}$ (with $\mathcal{M}_{Int} \preceq \mathcal{M}_{Index}$) is slightly smaller and, thus, it might be easier for humans to identify all relevant features of the feature-model composition. This benefit increases even more if more than one feature model is used for the feature-model composition (see Section 6).

Second, we took the scalability problem with feature-model analyses into account. Here, we focus on a scenario, where we are interested in the analysis of atomic sets for the feature model $\mathcal{M}_{DBMS/Int}$. If we use state-of-the-art implementations of this analysis, we perform the analysis for feature model $\mathcal{M}_{DBMS/Index}$ and finally filter the results to receive atomic sets that only contain features of interest (i.e., $\mathcal{F}_{DBMS/Int}$). By contrast, applying our concept of feature-model interfaces, we can use the feature model $\mathcal{M}_{DBMS/Int}$ for the analysis of atomic sets. Thus, we achieve the same results as in the state-of-the-art approach but with performance benefits due to the reduced propositional formula of feature model $\mathcal{M}_{DBMS/Int}$ compared to $\mathcal{M}_{DBMS/Index}$.

Third, we identified the support of feature-model evolution as one of our main challenges. Usually, an evolutionary change of feature model $\mathcal{M}_{Index}$ implies a complete re-computation of atomic sets for $\mathcal{M}_{DBMS/Int}$. By contrast, if we use feature-model interfaces, we only have to re-compute the analysis if the feature-model interface $\mathcal{M}_{Int}$ is no longer compatible with the evolved feature model $\mathcal{M}_{Index}$ (i.e., $\mathcal{M}_{Int} \not\preceq \mathcal{M}_{Index}$). To check this compatibility, we only need to compute the interface of the evolved feature model $\mathcal{M}_{Index}$ and to compare it to the previous interface version.

6. COMPOSITIONALITY IN THE WILD

Next, we explore feature-model interfaces in practice with thousands of features as given in industrial cases [11]. We investigate the typical size of interfaces and their potential to support humans and machines during evolution. Furthermore, we examine how often feature-model interfaces become incompatible to their corresponding, evolved feature models. In detail, we investigate the research questions:

RQ1: *How small can feature-model interfaces get compared to their corresponding feature models?*

RQ2: *How often does a feature-model interface become incompatible to an evolved feature model?*

Additionally, using the analysis of atomic sets, as it is the most computationally intensive analysis of our considerations with exponential complexity [18], we give an outlook on potential performance benefits of compositional analysis. Therefore, we investigate the following question:

RQ3: *Is it possible to achieve performance benefits using compositional analysis for atomic sets compared to an analysis of the complete feature model?*

6.1 Experimental Design and Subject

In our experiment, we investigate four monthly snapshots of one real-world feature model from the automotive domain, which includes features of hardware and software. We received it from our industrial partner in an obfuscated way (i.e., feature names are replaced by unique IDs). In snapshot V1, the feature model consists of 14,010 features with 666 constraints, whereas snapshot V4 has 18,616 features with 1,369 constraints (the feature models are available in the *Example Wizard* of FeatureIDE [20]).

The complete feature model of a snapshot originally contained more than 40 smaller feature-model instances. For our evaluation, we need the original feature-model instances because we want to investigate the relations to their feature-model interfaces. Fortunately, we are able to decompose the complete model into one root model and depending feature-model instances since we know the position of each instance in the complete feature model (i.e., the ID of its root feature). Regarding cross-tree constraints of the complete feature model, we distinguish between *intra-model constraints*, which describe dependencies within an instantiated feature model, and *inter-model constraints*, which describe dependencies between different models. We insert intra-model constraints in the corresponding feature model, whereas we save inter-model constraints for later usage.

For the evaluation, we use each instantiated feature model as input for our interface-generation algorithm and search for a strategy to select relevant features. Due to the lack of specific domain knowledge, we declared each feature that is included in an inter-model constraint as relevant (most notably, the root feature, due to its parent-child relationship to the root model). We call the resulting interfaces minimal because they only consist of features that are relevant for the composition (i.e., they can differ from interfaces designed by domain experts). Afterwards, we reconstruct a reduced feature model of the specific snapshot by recomposing the minimal feature-model interfaces and the root feature model. We perform this procedure for each snapshot and use the results to answer RQ1 and RQ2.

In order to use feature-model interfaces for this evaluation, we need a scalable generation algorithm. Although an algorithm of our previous work is suitable for the elimination of features in propositional formulas [48], the algorithm does not scale for the generation of feature-model interfaces. Therefore, we designed a new algorithm that is based on multiple satisfiability tests and logical resolution. However, the algorithm itself is out of our scope and discussed elsewhere in detail [26]. We refer interested readers to our open-source implementation in FeatureIDE v3.0 [20, 47].

To exemplify the compositionality properties of feature-model interfaces and to answer RQ3, we investigate the analysis of atomic sets and the results for the reduced feature model of snapshot V1. Contrary to the proposed atomic-set algorithm that only combines features with their mandatory children [42, 53], we use our Definition 2 of atomic sets, which may also combine features of different sub trees. For the computation of atomic sets for the reduced feature model, we consider two ways: (1) using the complete model with a subsequent filtering on the features of interest, and (2) using our recomposed feature model from the minimal interfaces. Afterwards, we compare the individual computation times.

6.2 Results and Discussion

We divide this section according to our research questions.

RQ1. In Figure 4, we depict the results of our investigation for research question RQ1 using boxplots (please ignore the bars for now). Each boxplot illustrates one snapshot of the automotive feature model and presents the percentage of features given in the interface relative to the features in the corresponding feature model. To further improve the illustration, we removed feature models with only one feature. For instance, in snapshot V1 there exists an instantiated feature model with exactly 7,800 features whereas the corresponding interface only consists of the root feature. The median of boxplot V1 and V2 is less than 2%, whereas the boxplots of snapshot V3 and V4 have a median less than 4%. Thus, half of all existing feature-model interfaces only consist of less than 4% of features relative to the corresponding feature models. Furthermore, 94% of all feature-model interfaces consist of less than 20% of the features.

In summary, the difference between the number of features in a feature-model interface and its corresponding feature model is significant. In most cases, the resulting feature-model interface consists of less than 20% of the features.

RQ2. For research question RQ2, we investigate the benefit of feature-model interfaces for an evolutionary scenario. In detail, we check how often adaptations of a feature-model interface are necessary, due to evolution of its corresponding feature model. In Figure 4, we depict the result of this investigation using bar charts that present the percentage of incompatible feature-model interfaces between all snapshots. For instance, the first bar (i.e., $\text{V1} \rightarrow \text{V2}$) presents whether the feature-model interfaces of snapshot V1 are still compatible with the feature models of snapshot V2 ($\mathcal{M}_{IntV1} \preceq \mathcal{M}_{V2}$). The results can be divided into three categories: (a) the feature-model interface is still compatible (green), (b) the interface is incompatible (red), and (c) the desired, minimal interface changed (yellow). In the last case, the new feature model is incompatible to its interfaces because the interface has a changed feature set due to new inter-model constraints. However, the dependencies within the feature model did not change and all features that are necessary to describe the inter-model constraints are available in the previous snapshot. Thus, they could be used to create the same interface as given in the current snapshot. Therefore, if a domain engineer creates ideal interfaces (i.e., using the knowledge of all future dependencies) instead of the minimal ones, the result would be a compatible feature-model interface for both snapshots.

In the bar $\text{V1} \rightarrow \text{V2}$, we present all 19 cases in which the feature model has been changed. Here, more than 84% of all feature-model interfaces are equal in both snapshots, less than 6% are not equal, and more than 10% are not equal because of minimality. In the other bars, only 14 and 13 feature models have been changed. As result, bar $\text{V2} \rightarrow \text{V3}$ presents more than 42% not equal feature-model interfaces, whereas the bar $\text{V3} \rightarrow \text{V4}$ again presents less than 8%. The equality of the new feature-model interface was decreased from 84% in bar $\text{V1} \rightarrow \text{V2}$ over 50% in $\text{V2} \rightarrow \text{V3}$ to 23% in bar $\text{V3} \rightarrow \text{V4}$. By contrast, the results from category (c) increase from 7% to 69% and, thus, the success of interfaces depends on the choice of removed features.

For research question RQ2, we investigated the interface dependency of evolved feature models to the interfaces of the

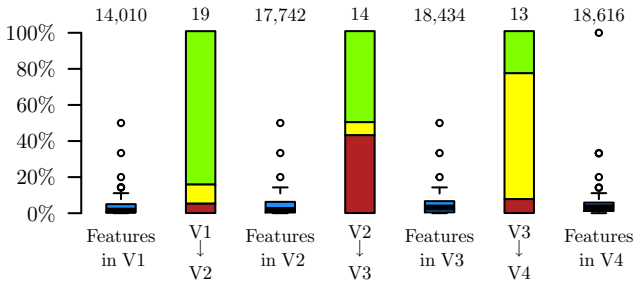

Figure 4: Percentage of features in the minimal feature-model interface compared to their feature models (■) and percentage of compatible interfaces (■), incompatible interface because of minimality (■), and incompatible interfaces (■).

previous version. We get the result that, in more than half of all cases in which the feature model has been changed, the interface dependency holds. For all positive cases, it is not necessary to change anything in the composed feature model and, thus, we need no further computations.

RQ3. Using the analysis atomic sets, we evaluate the potential of feature-model interfaces for compositional analyses. As described in our experimental design, we use the feature model of snapshot V1 as input and are interested in the atomic sets of the corresponding reduced feature model. The analysis based on the complete feature model takes more than 50 hours for the computation of all atomic sets, while the subsequent filtering is negligible. By contrast, in the composed analysis, we need in total less than 5 seconds for the generation of feature-model interfaces, reconstruction of the feature model, and computation of all atomic sets. Indeed, it is also possible to optimize the internals of the atomic sets algorithm to only consider relevant features. However, these optimizations are out of scope of this paper. Hence, we considered the algorithm as black box and used the described evaluation strategy. In summary, using the analysis of atomic sets, we illustrated that it is possible to reduce computational time using our concept of compositional analyses.

6.3 Threats to Validity

External Validity. The results of our study strongly depend on the analyzed feature model of the automotive domain, the distribution of the root features of the instantiated models, and on the features of interest ($\mathcal{F}_{Int}$) that we declared based on inter-model constraints due to the lack of domain knowledge. We plan to investigate whether the results can be generalized to other snapshots, feature models, or domains. Nevertheless, we had no influence on the selection of snapshots and root features we received from our industrial partner.

The snapshots of the automotive feature model seems to be extracted from an early state of the development process (i.e., snapshot V1 with 14,010 features and V4 with 18,616 features). It is possible that a more stable version leads to different results regarding necessary interface changes. However, we plan to investigate more snapshots and other case studies to get more insights.

Furthermore, we automatically generated minimal interfaces. In practice, developers will be able to incorporate domain knowledge to create interfaces. However, we received the feature model in an obfuscated way, which does not allow us to use domain knowledge for the interface generation.

Internal Validity. We used a prototypical implementation of our optimized algorithm to compute feature-model interfaces in a scalable manner. To reduce the probability of errors, we used unit tests in which the results of this algorithm were compared to the state-of-the-art algorithm for abstract features [48]. Because of scalability problems, it was not possible to compare the results of both algorithms with huge feature models but we used smaller feature models for the comparison with different sets of relevant features.

With refactorings, it is possible to further reduce our minimal feature-model interfaces. For instance, let us consider the constraint $A \Rightarrow B \wedge C$ as inter-model constraint, in which the features A and B are features of the same instantiated feature model and C is part of another feature model. Here, we include both features in the corresponding feature-model interface. However, it is possible to refactor this constraint into two new constraints $A \Rightarrow B$ as intra-model constraint and $A \Rightarrow C$ as inter-model constraint. As result, it is sufficient to only include A into the feature-model interface instead of A and B. This approach could lead to even better results but may harm readability.

7. RELATED WORK

Here, we present several works in the domains of interfaces in product lines, feature-model composition, the analysis of feature models, and further analyses based on interfaces.

Interfaces and Views for Product Lines. As stated above, the definition of the slice operator presented by Acher et al. is similar to our definition of feature-model interfaces [4]. Thus, the slice operator uses a feature model as input to create a new one that only consists of a subset of features with unchanged feature dependencies. However, Acher et al. use the slice operator to focus on the property of feature-model decomposition. In subsequent work, Acher et al. use the slice operator in combination with a merge operator to consider evolutionary changes of extracted variability models in a real-world plugin system [2]. For the extraction process, the authors used feature-model aggregation and slicing and compare the different models of each system's version. Whereas we aim to support evolution using a stable feature-model interface so that we prevent a re-evaluation of the system, Acher et al. aim to detect differences of the feature-model versions during evolution.

Furthermore, Dhungana et al. present an interface that is mainly used for information hiding to other parts of the feature model (called fragments) and to support evolution [17]. In detail, the authors save a merge history of fragments at one point in time, give feedback to the single fragments to ease their maintenance and use the history to re-merge the fragments in the future. However, the approach does not consider automated analyses.

A further concept related to feature-model interfaces are feature-model views [23, 31, 37]. Views also present a subset of relevant features based on a master feature model and are generally used to ease the configuration of large scale feature models. Thus, different views regarding one master feature model are combined to get a valid configuration based on the view's partial configurations. By contrast, a feature-model interface can be an interface of a set of different feature models and is not bound to a specific one.

Automated Analyses of Feature Models. There exists a wide range of research for automated analyses. Benavides et al. present a survey about existing analyses of feature models with information regarding the analysis concept, tool support, and references to work on particular analyses [9]. Based on this, Durán et al. present a formal framework in which these analyses can be described [18]. However, the necessity for automated analyses of feature models was introduced together with feature models themselves. Kang et al. already recognize that tool support is essential to create accurate feature models for complex domains [24]. Therefore, they propose a tool based on Prolog using a fact base and composition rules [24]. By contrast, todays tools are typically based on satisfiability solvers or binary decision diagrams [10, 32, 43, 47]. While the check for satisfiability of a propositional formula is an NP-complete problem, Mendonca et al. claim that satisfiability checks in the domain of feature models scale well in most cases [33]. In contrast, the computations of some other analyses, such as atomic sets, do not scale for large feature models.

Feature-Model Composition. Composition mechanisms are often used for multi software product lines that are a set of multiple dependent product lines [22, 27, 34]. In this context, large-scale variability modeling is essential to fulfill the system's requirements. Eichelberger and Schmid give an overview of textual-modeling languages that can be used for large-scale variability modeling [19]. The authors identify several languages, such as, FAMILIAR [5], VELVET [35], TVL [13], and VSL [1], which support variability-model composition and compare the languages regarding their facility to support *composition*, *modularity*, and *evolution*. In general, it is also possible to integrate our approach into other languages to facilitate compositional analyses. Furthermore, as integrated in the language and tool FAMILIAR, Acher et al. compare a set of composition operators, such as a merge operator based on union and intersection of feature sets [6]. In detail, the authors investigate different implementation options and present advantages and drawbacks. For our investigation of compositional analyses, we relied on an own formal description inspired by the aggregation mechanism of the modeling language VELVET [35]. However, the formal definition of our composition mechanism can be regarded as a combination of the aggregation and the merge operator introduced in FAMILIAR [5].

Product-Line Analyses Based on Interfaces. It is an open question how much the different kinds of product-line analyses, such as family-based analysis of product lines [46], can benefit from using feature-model interfaces. However, in our previous work, we presented an overall concept of interacting interfaces [39], in which the feature-model interfaces represent the central part for the subsequent interfaces. In detail, we also introduced *syntactical* and *behavioral* product-line interfaces. Syntactical interfaces support users during the implementation of the product line and present a view on reusable programming artifacts [40]. By contrast, behavioral interfaces are used to ease the product-line verification [49]. Besides our own investigations, Kästner et al. introduced a variability-aware module system that allows for type checking modules in isolation. For this purpose, the authors define interfaces between these modules, in which also the variability is encoded [25]. However, these interfaces do not hide variability as it is focused by feature-model interfaces. Furthermore, Li et al. also present interfaces for feature-oriented systems to verify the product-line behavior [29]. For the formal representation of these systems, the authors use state machines as input for the verification of properties that they describe in temporal logic. To ease the verification and to facilitate a feature-based verification, interfaces are used to encapsulate the connection states to other feature's state machines. This is similar to our consideration of minimal interfaces in which we only considered features that are involved in an inter-model constraint.

8. CONCLUSION

Highly-configurable systems can have thousands of options and each option may have dependencies to other options. These dependencies are typically specified by means of feature models. However, experiences with applying our tool FeatureIDE in industry is that large feature models are overwhelming and hard-to-understand for all stakeholders. Furthermore, large feature models often slow down analyses for configurable programs as they typically involve satisfiability problems with a variable for each option. This situation gets even worse during evolution; stakeholders and analyses need to consider the complete model when reasoning about the impact of a change even though it may have only local effects. While there are some recent attempts to decompose feature models, they still require the composition into a large model prior to analysis.

We propose feature-model interfaces for composition of feature models to establish information hiding. A feature-model interface is a feature model that represents a subset of features that are relevant to certain stakeholders, and is used for composition with other feature models. On the one hand, we proved compositionality properties for several analyses, i.e., we state under which circumstances a change does not affect other parts. On the other hand, we empirically investigated a real-world feature model from the automotive domain with 18,616 features and its three-month-history, for which a decomposition was already available. In the majority of all cases, feature-model interfaces may contain less than 4% of the features. Furthermore, changes to the hidden feature models do not require changes to the according interfaces in more than half of the cases. For a particular analysis, we measured a decrease in computation time.

While our results make us confident that the compositionality properties of feature-model interfaces have a positive effect on numerous existing analyses for configurable programs, we are still at the beginning of a long highway requiring further efforts in both, theory and empirical evaluation. We are currently working on further analyses and an empirical evaluation with the Linux kernel.

ACKNOWLEDGMENTS This work is partially funded by BMBF grant (01IS14017B), DFG grant (SCHA1635/4-1) and by the European Commission (ERC H2020-644298). For the snapshots of the real-world feature model, we thank our industrial partner. We also thank the participants of the yearly FOSD meeting, in particular Christian Kästner, Sven Apel, Christoph Seidl and the reviewers of this paper for their constructive feedback. An early draft of this paper is also available as technical report [38].

9. REFERENCES

[1] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren, and M. Weber. The CVM Framework - A Prototype Tool for Compositional Variability Management. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 101–105. Universität Duisburg-Essen, 2010.

[2] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and Evolution of Architectural Variability Models in Plugin-Based Systems. *Software and System Modeling (SoSyM)*, 13(4):1367–1394, 2014.

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

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

[5] M. Acher, P. Collet, P. Lahire, and R. B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)*, 78(6):657 – 681, 2013.

[6] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France. Composing Your Compositions of Variability Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 352–369. Springer, 2013.

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

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

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

[10] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134. Technical Report 2007-01, Lero, 2007.

[11] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013.

[12] M. Bošković, G. Mussbacher, E. Bagheri, D. Amyot, D. Gašević, and M. Hatala. Aspect-Oriented Feature Models. In *Proc. Int'l Conf. Models in Software Engineering (MODELSWARD)*, pages 110–124. Springer, 2011.

[13] A. Classen, Q. Boucher, and P. Heymans. A Text-based Approach to Feature Modelling: Syntax and Semantics of TVL. *Science of Computer Programming (SCP)*, 76(12):1130–1143, 2011.

[14] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

[15] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[16] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006.

[17] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *Journal of Systems and Software (JSS)*, 83(7):1108–1122, 2010.

[18] A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FLAME: A Formal Framework for the Automated Analysis of Software Product Lines Validated by Automated Specification Testing. *Software and System Modeling (SoSyM)*, pages 1–34, 2015. In Press.

[19] H. Eichelberger and K. Schmid. A Systematic Analysis of Textual Variability Modeling Languages. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 12–21. ACM, 2013.

[20] FeatureIDE Development Team. FeatureIDE GitHub, 2015. https://github.com/FeatureIDE/FeatureIDE.

[21] A. Hemakumar. Finding Contradictions in Feature Models. In *Proc. Int'l Workshop on Analyses of Software Product Lines (ASPL)*, pages 183–190. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[22] G. Holl, P. Grünbacher, and R. Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *J. Information and Software Technology (IST)*, 54(8):828–852, 2012.

[23] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards Multi-View Feature-Based Configuration. In *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 106–112. Springer, 2010.

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

[25] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792. ACM, 2012.

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

[27] C. W. Krueger. New Methods in Software Product Line Development. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 95–102. IEEE, 2006.

[28] K. Lauenroth, K. Pohl, and S. Toehning. Model

Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[29] H. Li, S. Krishnamurthi, and K. Fisler. Interfaces for Modular Feature Verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 195–204. IEEE, 2002.

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

[31] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-Oriented Variability Modeling. In *Proc. Computer Software and Applications Conf. (COMPSAC)*, pages 67–72. IEEE, 2009.

[32] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762. ACM, 2009.

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

[34] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. Universität Duisburg-Essen, 2010.

[35] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-Dimensional Variability Modeling. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–22. ACM, 2011.

[36] M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and C. Kästner. Modeling Dependent Software Product Lines. In *Proc. Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 13–18. Department of Informatics and Mathematics, University of Passau, 2008.

[37] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 252–268. Springer, 2012.

[38] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. Feature-Model Interfaces for Compositional Analyses. Technical Report FIN-001-2015, University of Magdeburg, Germany, 2015.

[39] R. Schröter, N. Siegmund, and T. Thüm. Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Software Product Line Conference co-located Workshops*, pages 96–99. ACM, 2013.

[40] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 102–111. ACM, 2014.

[41] R. Schröter, T. Thüm, N. Siegmund, and G. Saake. Automated Analysis of Dependent Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9:1–9:5. ACM, 2013.

[42] S. Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Workshop on Analyses of Software Product Lines (ASPL)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[43] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 63–71. ACM, 2012.

[44] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012.

[45] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

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

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

[48] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200. IEEE, 2011.

[49] T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, and S. Krüger. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104. ACM, 2016.

[50] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software (JSS)*, 81(6):883–896, 2008.

[51] P. Trinidad, D. Benavides, and A. Ruiz-Cortés. Improving decision making in software product lines product plan management. In *Proc. Workshop on Decision Support in Software Engineering (ADIS)*, pages 1–8. RWTH Aachen, 2004.

[52] P. Trinidad and A. Ruiz-Cortés. Abductive Reasoning and Automated Analysis of Feature Models: How are They Connected? In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 145–153. Universität Duisburg-Essen, 2009.

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