Automated Analysis of Dependent Feature Models

Reimar Schröter, Thomas Thüm, Norbert Siegmund, Gunter Saake*
University of Magdeburg
Magdeburg, Germany

ABSTRACT

Feature models specify valid combinations of features in software product lines. With dependent feature models (DFMs), we apply separation of concerns to feature models for two main benefits. First, we can modularize feature models into parts relevant to groups of stakeholders. Second, we are able to model dependencies between different software product lines in a multi-product-line scenario. To ensure consistency and correctness of DFMs, we have to apply analyses, such as dead-feature detection. We discuss why DFMs challenge the detection of inconsistencies, present how to reuse existing analyses for DFMs, and propose new analyses to supplement existing ones. We apply automated analyses in five steps and evaluate the approach using DFMs specified in VELVET by our prototype VeAnalyzer.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Design, Verification

1. INTRODUCTION

A *software product line (SPL)* is a set of similar programs (products) developed using a common code base [7]. *Features* describe prominent or distinctive user-visible aspects of these different programs [9]. To describe valid combinations of features (i.e., products), we can use feature models to define relationships and constraints among the features [2]. However, when constraints are not properly used, feature models can become inconsistent. Automated analysis can be used to detect inconsistencies, such as *dead features* which are included in no product, or a void feature model which represents no products [3].

*This work is partially funded by DFG grant SA 465/34-2, and BMBF grant 01IM10002B.

VaMoS '13, January 23 - 25 2013, Pisa, Italy

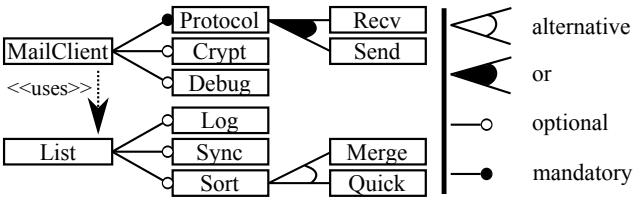

Figure 1: Feature models of a `List` and a `MailClient`.

There are two reasons to modularize feature models. First, feature models in industry tend to get very large (e.g., up to 11 000 features [15]) and stakeholders may only be experts for certain parts of a feature model. Separating feature models is a powerful technique to deal with a growing complexity. Second, with an increasing development of SPLs in recent years, products of an SPL are used not only stand-alone, but also within products of other SPLs. A simple approach to reuse an SPL is to configure and derive a valid product, which can be used in other SPLs. However, this approach is insufficient when different products of the same SPL are needed depending on the configuration of the reusing SPL. The second scenario are *multi software product lines (MPLs)*, which are arbitrary compositions of interdependent SPLs [11]. For example, we can create an MPL `MailClient` that needs different configurations of SPL `List`, such as a sorted list to store mails and a synchronized list to log errors (see Figure 1).

Separation of complex feature models and the reuse of existing ones results in two application scenarios for *dependent feature models (DFMs)*; we define DFMs as follows:

Definition 1 (Dependent feature model): *A dependent feature model is an extension of a feature model that permits any feature modeling construct of a given feature modeling notation (e.g., or-group, alternative-group, optional features, mandatory features, constraints), but may also contains references or dependencies to other (dependent) feature models.*

Related to our definition, feature models are also DFMs. Additionally, DFMs can contain references or dependencies to other DFMs. For example, the modeling languages TVL and VELVET support such complex model descriptions that we call DFMs [6, 12].

DFMs challenge existing techniques for inconsistency detection. Analyzing each DFM in isolation (separate analyses of feature models `List` & `MailClient`) is often not applicable because these models may have dependencies to other DFMs. To overcome this problem, we propose means to analyze DFMs. Specifically, we make the following contri-

butions: we (1) present challenges by DFMs that complicate the application of existing analyses, (2) introduce a five-step approach to address these challenges, (3) propose a transformation of DFMs into a feature model to enable reuse of existing analyses, and (4) introduce new analyses for DFMs.

2. DEPENDENT FEATURE MODELS

In Figure 2, we present a UML diagram illustrating the dependencies between SPL `MailClient` and SPL `List` [13]. SPL `MailClient` uses two different products (i.e., instances) of SPL `List` (`SyncList` & `SortList`). We use VELVET — a feature modeling and configuration language that supports separation of concerns [12] — to define DFMs. Our results are not specific to VELVET and can be applied to other feature modeling notations.

In Listing 1-(a), we use SPL `List` to illustrate how to use VELVET. An SPL is described by a `concept`, in which optional features of an SPL are defined with keyword `feature`. The additional keyword `mandatory` can be used to define obligatory features. Furthermore, we define alternative-groups with keyword `oneof`, such as features `Merge` and `Quick`, and an or-group with keyword `someof`. We use keyword `constraint` to define cross-tree constraints between features with logical operators `&&` (conjunction), `||` (disjunction), `!` (negation), `->` (implication), and `<->` (biconditional) [12].

VELVET is inspired by object-oriented programming languages allowing us to use mechanisms of aggregation and inheritance to compose DFMs. Although VELVET supports superimposition as another composition mechanism, we focus on inheritance and aggregation for brevity.

Inheritance of feature models is a mechanism, in which the inherited feature model contains all features and constraints of the super feature model plus additional features and constraints. For example, DFM `SyncList` (cf. Figure 2 & Listing 1-(b)) can be defined as all features and constraints of feature model `List` and a further constraint that reduces the number of possible products in the DFM `SyncList` (see Listing 1-(b)). Thüm et al. call this reduction of possible products a *specialization* [16]. But inheritance even permits to add features to the existing ones and thus to produce more products, which is called *generalization* [16]. However, VELVET also allows us to add features and constraints simultaneously, which often results in an *arbitrary edit*.

In VELVET, *aggregation* allows us to use feature models in another feature model. We define aggregation with a concept and instance name. This specific characteristic of DFMs gives us the opportunity to define constraints related to this specific instance without an influence to other instances of the same concept. For example, we use the concepts `SyncList`, `SortList` and the identifiers `mailList`, `errList` as instances in the DFM `MailClient` (see Listing 1-(c)). Furthermore, we add constraint `mailList.Sort.Merge` to specify that instance `mailList` uses algorithm merge sort.

Challenges. The expressiveness by the use of mechanisms such as inheritance and aggregation of DFMs results in the following challenges related to automated analysis.

C1: We cannot directly apply state-of-the-art operations for automated analysis, because of *separation of concerns* in DFMs. For example, the complete model of the `MailClient` is separated over different (fragments of) feature models. The single concepts cannot be analyzed separately, because not all definitions are included in one concept such that there can be reference errors (e.g., features are referenced that not exist in this concept). For example, if we analyze the DFM `SortList`, only one constraint is defined in the corresponding concept.

Figure 2: Dependencies between different SPLs [13].

(a) Feature model

```
concept List {
  feature Sync;
  feature Sort {
    oneof { feature Merge;
            feature Quick; } }
  feature Log; }
```

(b) Inheritance

```
concept SyncList : List {
  constraint List.Sync; /*Specialization*/ }
concept SortList : List {
  constraint List.Sort;
  feature List.Sort.Bubble; /*General.*/ }
```

(c) Aggregation

```
concept MailClient {
  SortList mailList;
  constraint mailList.Sort.Merge;
  SyncList errList; /*...*/ }
```

Listing 1: Definition of DFM in VELVET.

C2: VELVET includes different *composition mechanisms*, such as inheritance and aggregation, which must be correctly combined to analyze DFMs. Thus, the challenge is to express the semantics of different composition mechanisms (e.g., instances can only be specialized, whereas inheritance provides arbitrary edits of based models) to transform DFMs into a representation that can be analyzed.

C3: Due to the separated definitions of DFMs in different concepts, we face further *syntactical problems*. For example, we must check whether features and constraints are correctly defined, such as whether feature `Sort` exists in the concept of `SortList` or `List`. This is more complex as in standard feature models, because the referenced features may be defined in any other DFM that is referenced.

3. THE FIVE-STEP APPROACH

In Figure 3, we present our five-step approach for automated analysis of DFMs. Our concept bases on a two-step approach in which (1) the feature model is transformed to a logical format and (2) a solver is used to perform analyses (e.g., satisfiability solver) [3]. Using VELVET's analogy to object-oriented programming, we can consider our steps as (S1) source code parsing, (S2) type checking, (S3) compilation, and (S4 & S5) execution. Here, we describe steps S1-S4 and focus on analyses of step S5 in Section 4.

Step S1: Model Parsing. For the translation of feature models into propositional formulas, a tool has to parse a textual representation of feature models. Parsing in context of DFMs means that we detect errors, which are related to an incorrect usage of the grammar (in our case, the VELVET grammar). For example, in VELVET it is not allowed to define a mandatory feature in an alternative-group, because all siblings would be dead features. Hence, we may detect dead features simply by checking whether a concept is valid according to the VELVET grammar.

Contrary to the initial version of VELVET [12], we pro-

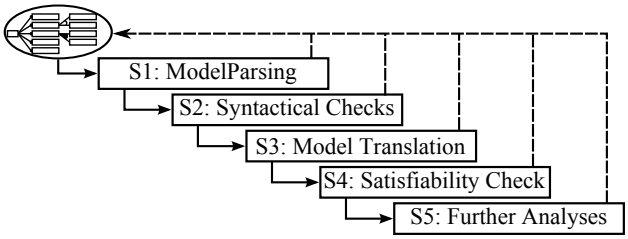

Figure 3: Five-step approach of automated analyses.

pose to restrict the grammar to allow not more than one group of sub-features. Otherwise by the use of inheritance, we cannot automatically decide in which group the feature should be added. For example, if we define another group next to the alternative-group of feature `Sort` (see Listing 1-(a)), we are not able to decide automatically which group is extended by feature `Bubble` (see Listing 1-(b)).

Step S2: Syntactical Checks. Similar to type checking for object-oriented languages, we must check the correctness of feature references. Here, we present some examples.

We cannot detect missing or wrong references with parsers and, thus, we must check whether referenced features (e.g., a feature in a constraint) are declared. In Listing 1-(b), we define concept `SyncList` and inherit the features and constraints of SPL `List` (see Listing 1-(a)). As result, we must investigate concept `SyncList` whether feature `Sync` exists and if not, continue in the inheritance hierarchy to check whether SPL `List` contains feature `Sync`.

Step S1 is not sufficient to detect mandatory features, because a mandatory feature may be added to a group defined in another concept. For example, if we define a new feature in an inherited concept (see feature `Bubble`, Listing 1-(b)), it is not sufficient anymore to parse only a single feature model to check for the keyword mandatory.

Step S3: Model Translation. If we analyze a DFM with existing analysis operations for feature models, we need a single propositional formula that represents all features and dependencies of each part of the DFM. To this end, we merge the dependent VELVET concepts into one feature model by considering the specific composition mechanisms (e.g., inheritance & aggregation). Afterwards, we translate the composed feature model into one propositional formula.

When composing two DFMs with inheritance, we use the base concept as starting point. Next, we add all new definitions (e.g., features & constraints) to this existing feature model and rename the root feature to the concept name of the inherited concept. In Figure 4, we present the transformation of DFM `SortList`, which consists of two concepts (`SortList` & `List`) that are connected via inheritance. On top, we show the feature model of SPL `List` from which DFM `SortList` inherits all features and constraints. Afterwards, we add all features that are defined in concept `SortList` and add the additional cross-tree constraints.

We transform DFMs with aggregation as follows. If an instance is defined, we use the type of the instance (the concept), rename the root to the instance name and transform the instance root to a mandatory sub-feature of the feature in which the instance is defined. Renaming of the root feature is important because several instances of the same type may occur below the same feature. For example, we present the transformation of DFM `MailClient` with two instances of type `List` in Figure 4. We integrate the complete feature

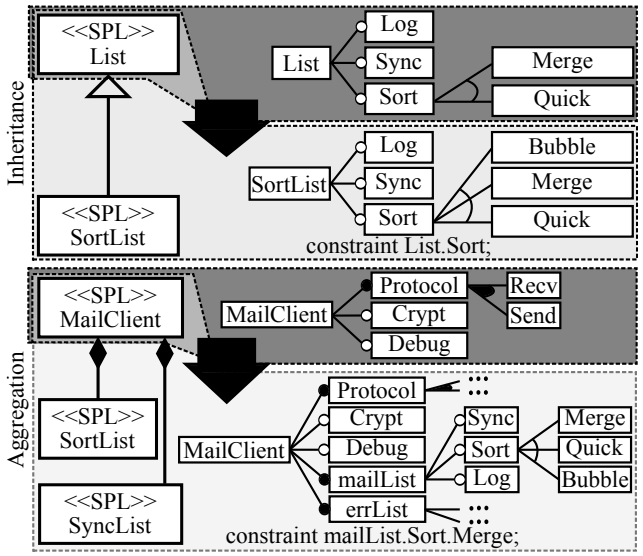

Figure 4: Transformation of DFMs in VELVET.

model of each instance (DFM of `SortList` and `SynList`, which is composed by inheritance based on `List`), rename the root feature to the instance identifier (`mailList` & `errList`), and connect these roots as mandatory to feature `MailClient`.

Step S4: Satisfiability Check. We separate the check for void feature models from other analyses, because we argue that this analysis represents a precondition for the following analyses. The reason is that by applying the void feature model analysis first, we can detect inconsistencies of the feature model at an early stage such that we can cancel further analysis and save unnecessary computations. The analysis of dead features is one example in which the satisfiability check is a precondition that can save computation effort because a void feature model has only dead features. Hence, instead of performing a dead-feature analysis per feature, we perform the satisfiability analysis and abort the analysis process if the feature model is void.

4. ANALYSIS OF DFMS

Next, we present analyses that can be executed based on propositional formulas and investigate whether these analyses produce correct results in the context of DFMs.

Dependent-dead feature. A feature is called dead feature if it is not included in any product of an SPL [3]. When determining dead features of the DFM `ExtList` (see Listing 2), which was translated to one feature model (Step S3), we get features `Quick` and `Bubble` as dead features. However, related to the definition of an anomaly, this seems contradictory. In feature model `List` exists no dead feature and in `ExtList`, we purposefully selected feature `Merge` so that feature `Quick` is not a real anomaly but a desired configuration decision (i.e., for a DFM we may have bind some variability of another DFM). To overcome this contradiction, we propose the following definition:

Definition 2 (Dependent-dead feature): *A feature is a dependent-dead feature, if it is not included in any product of that DFM, in which it is declared.*

If we use the new definition of dependent-dead features

```
1 concept ExtList : List {
2   feature List.Sort.Bubble; //added to group
3   constraint List.Sort.Merge; }
```

Listing 2: DFM `ExtList`.

related to DFM `ExtList`, we observe that only feature `Bubble` is a dependent-dead feature. This is our desired result because feature `Bubble` is defined in concept `ExtList` and should be at least selectable in this concept, but this feature can neither be used in feature model `List` nor in `ExtList` and is thus an inconsistency.

Dependent-false-optional feature. A false-optional feature is a feature that is part of each SPL's product, although it is not designed as mandatory [3]. We want to detect only those features as false-optional features that are actual anomalies. If we determine false-optional features of `ExtList`, we get the false-optional features `Merge` and `Sort` as result. However, considering the constraints in Listing 2, we purposefully want to reduce the number of products so that these features are not considered as an anomaly. We conclude that also this definition must be adapted. We obtain the desired result (an empty set of dependent-false-optional features) by the use of the following definition:

Definition 3 (Dependent-false-optional feature): *A feature is a dependent-false-optional feature, if it is not designed as mandatory, but is part of all products of the DFM in which the feature is defined.*

Core features and core instances. Benavides et al. consider core features as the most important features that must be implemented first because they are included in every product [3]. When we determine core features of DFM `ExtList`, we yield features `ExtList`, `Sort`, and `Merge` as a result. By contrast, in feature model `List` exists only the core feature `List`.

Instead of analyzing the whole DFM in one step, we can take advantage of the known structure of the DFM by analyzing the feature model in a stepwise manner. For example, we compute that features `Merge` and `Sort` are new core features of concept `ExtList`, whereas feature `List` (resp. feature `ExtList`) was already a core feature in the inherited concept. Thus, we can check the impact of the newly defined elements in extensions of feature models using inheritance.

We observed that the number of core features can be large. However, sometimes we may want to know only the set of instances that are included in all products of a DFM, which represents a subset of the complete set of core features. For example, the analysis whether instances `mailList` and `errList` are instances that are included in all products of the MPL `MailClient` could be sufficient for a user. Thus, we propose a new analysis to calculate core instances (both instances are core instances in our example):

Definition 4 (Core instance): *An instance of a DFM is a core instance if the root feature is a core feature of the DFM.*

Number of products and variability of instances. Another commonly used analysis calculates the number of valid products [3]. With it, we can get an impression about the complexity of feature models and the set of possible products. For example, if we determine number of products of DFM `ExtList`, we get four possible products as result. Similar to the previous analysis, we can use relations between

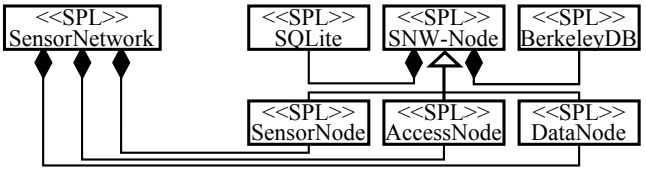

Figure 5: Relationships of sensor networks [11].

DFM `ExtList` and other DFMs (e.g., `List`) to determine that, in this example, inheritance reduces the number of products from twelve to four products of `ExtList`. By comparing these results, we can detect possible errors.

Additionally, if a user defines an instance, it can be useful to know how many products are represented by this instance after the configuration in DFMs. Instance `mailList` represents four possible configurations, and thus, `mailList` is not fully specialized and can be configured by a user. This is not desired in all scenarios and, thus, it is important to know the variability of instances.

To determine number of products related to an instance, it is not sufficient to use the type of an instance as input model for the analysis number of products. Because, this analysis neglect possible configurations in a DFM which must be considered. For example, the number of products of DFM `SortList` is twelve whereas the number of remaining configuration options of the instance in the DFM `mailList` is four. We propose the following algorithm to compute the latter number. We investigate all features of the instance `mailList` related to the complete DFM `MailClient`. For that reason, we use the complete DFM as propositional formula and remove all features, which are not sub-features of the instance `mailList`. Afterwards, the propositional formula includes only features of instance `mailList` with all constraints that were defined in the complete DFM of MPL `MailClient`. Thüm et al. presented an algorithm to remove features from a propositional formula, which we can use in our scenario [17]. Afterwards, we determine the number of products of this new feature model (i.e., the represented instance with defined constraints of the `MailClient`). As a result, we compute four possible products.

5. EXPERIENCES WITH VEANALYZER

We developed the prototype VeAnalyzer,[1] a command line tool for automated analysis of DFMs specified in VELVET. We used VeAnalyzer to analyze the DFM `SensorNetwork` (see Figure 5), which consists of multiple real-world feature models (`SQLite` & `BerkeleyDB`) and simulates a sensor network. In detail, we used each DFM as input for VeAnalyzer and compared the results with our manual investigation. Syntactical errors caused by a new VELVET-grammar version were detected and could be repaired (step S1 & S2). Afterwards, the DFMs were transformed to a propositional formula and the analyses were executed (see Table 1).

In detail, we determined the number of features and all instances of the complete DFMs to give an impression about the DFM complexity. Furthermore, column *satisfiable* illustrates that all DFMs passed the satisfiability check (step S4), which is the basis for all subsequent analyses (step S5). Using our proposed analyses to detect anomalies, we identified that no dependent-dead features and dependent-false-

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/MultiPLe/modeling.htm`

DFM	f	i	sat	ddf	dfof	cf	cfi	ci
SQLite	86	0	✓	0	0	8		0
BerkeleyDB	14	0	✓	0	0	1		0
SnwNode	127	2	✓	0	0	3		0
SensorNode	131	2	✓	0	0	4	1	0
AccessNode	128	2	✓	0	0	15	12	0
DataNode	128	2	✓	0	0	16	13	1
SensorNetwork	393	9	✓	0	0	20		2

Legend: f – features, i – instances, sat – satisfiable, ddf – dependent-dead feature, dfof – dependent-false-optional feature, cf – core feature, cfi – new core feature through inheritance, ci – core instance

Table 1: Results of VeAnalyzer.

optional features exist. Furthermore, we determined core features in the complete DFM, whereas the next column gives an impression of the inheritance effects. For example, DFM `DataNode` consists of 16 core features whereas 13 features are new core features of concept `DataNode`. In addition, instance `SQLite` exists in each product of `DataNode` and thus, is a core instance. We conclude that 8 of the new 13 core features of `DataNode` are features of this core instance.

Additionally to the sensor-network example, we created unit tests to check whether our analyses and steps are correct. In detail, we created DFMs with dependent-dead features, dependent-false-optional features, and core instances to analyze them with VeAnalyzer. Once again, we compared the results of VeAnalyzer with our manual investigation and verified the correctness of the analyses.

6. RELATED WORK

This paper is based on the work of Rosenmüller et al. who introduced VELVET [12]. Here, we intend to complement the modeling techniques by automated analyses for DFMs. By contrast, our results are not specific to VELVET and can be applied to other feature modeling languages, such as the text-based feature modeling language TVL [6].

Besides these languages, which allows us to modularize feature models, several work exists about composing feature models [1, 5]. Similar to the composition mechanisms of VELVET, rules are defined to combine or merge feature models. For example, Acher et al. introduced special operators, such as insert and merge, that allow us to combine feature models similar to VELVET [1]. Boskovic et al. use aspect-oriented techniques to define parts of feature models that can be combined via composition rules [5]. These rules provides the possibility to apply automated analysis.

The separation of feature models in smaller ones is only one solution to manage complexity of feature models. By contrast, also views on parts of large feature models can be used to treat the complexity [8, 10, 14]. Configurations in such views related to different concerns of one feature model must be merged by specific rules and checked for consistency. Similar to this work, analyses are important to check for correctness of the composition of these specific concerns.

The use of a satisfiability solver with propositional formulas is not the best solution for all analysis operations, for example, we can determine the number of products by the use of BDDs more efficiently [4]. Our five-step approach and the proposed analysis operations are not specific to satisfiability solvers and can easily be used with other solvers.

7. CONCLUSIONS

Automated analyses are crucial to check the correctness of feature models. DFMs are an extension of feature models and yield new requirements that prohibit direct application of state-of-the-art analyses. To overcome this problem, we propose a five-step approach that allows us to analyze DFMs. Moreover, we introduce special analyses for DFMs, such as dependent-dead features, dependent-false-optional features, core instances, and instance variability.

We presented the prototype VeAnalyzer that supports our results. In future work, we want to integrate further analysis operations into VeAnalyzer, include further composition techniques of DFMs, and show how to apply our five-step approach to other modeling languages.

8. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Composing Feature Models. In *SLE*, page 62–81. Springer, 2009.

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

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[4] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *SPLC*, page 39–47. IEEE, 2006.

[5] M. Boskovic, G. Mussbacher, E. Bagheri, D. Amyot, D. Gasevic, and M. Hatala. Aspect-Oriented Feature Models. In *MoDELS*, page 110–124. Springer, 2010.

[6] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *VaMoS*, page 159–162. University of Duisburg-Essen, 2010.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[8] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards Multi-View Feature-Based Configuration. In *REFSQ*, page 106–112. Springer, 2010.

[9] 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.

[10] M. Mannion, J. Savolainen, and T. Asikainen. Viewpoint-Oriented Variability Modeling. In *COMPSAC*, page 67–72. IEEE, 2009.

[11] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *VaMoS*, page 123–130. University of Duisburg-Essen, 2010.

[12] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-Dimensional Variability Modeling. In *VaMoS*, page 11–22. ACM, 2011.

[13] M. Rosenmüller, N. Siegmund, S. S. ur Rahman, and C. Kästner. Modeling Dependent Software Product Lines. In *McGPLE*, page 13–18. University of Passau, 2008.

[14] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *MoDELS*, page 252–268. Springer, 2012.

[15] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. In *PLOS*, page 2:1–2:5. ACM, 2011.

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

[17] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, page 191–200. IEEE, 2011.