# Quantifying the Variability Mismatch Between Problem and Solution Space

Marc Hentze
marc.hentze1@volkswagen.de
Volkswagen AG
Germany

Chico Sundermann
chico.sundermann@uni-ulm.de
University of Ulm
Germany

Thomas Thüm
thomas.thuem@uni-ulm.de
University of Ulm
Germany

Ina Schaefer
ina.schaefer@kit.edu
Karlsruhe Institute of Technology
Germany

## ABSTRACT

A software product line allows to derive individual software products based on a configuration. As the number of configurations is an indicator for the general complexity of a software product line, automatic #SAT analyses have been proposed to provide this information. However, the number of configurations does not need to match the number of derivable products. Due to this mismatch, using the number of configurations to reason about the software complexity (i.e., the number of derivable products) of a software product line can lead to wrong assumptions during implementation and testing. How to compute the actual number of derivable products, however, is unknown. In this paper, we mitigate this problem and present a concept to derive a solution-space feature model which allows to reuse existing #SAT analyses for computing the number of derivable products of a software product line. We apply our concept to a total of 119 subsystems of three industrial software product lines. The results show that the derivation scales for real world software product lines and confirm the mismatch between the number of configurations and the number of products.

## CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; Embedded software; Model-driven software engineering.

## KEYWORDS

Product lines, variability mismatch, solution-space analyses

**ACM Reference Format:**
Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the Variability Mismatch Between Problem and Solution Space. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3550355.3552411


*MODELS '22, October 23–28, 2022, Montreal, QC, Canada*

ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552411

## 1 INTRODUCTION

Today, software-intensive systems are often configurable to meet the individual requirements of customers. Such systems can be managed as a software product line [15] which allows to automatically derive individual *products* for a given selection of *features*, called *configuration*. A software product line is split into *problem space* and *solution space* where the problem space defines available features and their dependencies and the solution space contains the *implementation artifacts* that realize the features on a technical level [15]. Both spaces are connected by the *configuration knowledge* that defines the mapping between features and their implementation artifacts. This allows to automatically select the implementation artifacts that are required to construct a specific product for a given configuration. In industry, software product lines commonly allow a high number of valid configurations which results in different challenges such as ensuring the functional correctness of each individual configuration [10, 39, 49]. Therefore, the number of configurations is a central indicator for the complexity of a software product line.

Since determining the number of configurations manually is infeasible, #SAT-based analyses have been proposed that compute the number of problem-space configurations automatically [42]. Those analyses are commonly tailored to variability models (i.e., feature models [8]) which are used to define the problem-space features and their dependencies of a software product line. The number of problem-space configurations, however, does not necessarily need to match the number of solution-space products. This is because common variability realization mechanisms, such as preprocessor directives [38, 41], often result in a feature implementation that is scattered across multiple artifacts or in an implementation artifact whose inclusion is controlled by multiple features. Due to the mismatch, using the number of configurations to reason about the software complexity, i.e. the number of products, is misleading. Since the software complexity is a central aspect when implementing and testing a software product line, it is highly beneficial to know the exact number of derivable products. Even though the mismatch between the number of problem-space configurations and the number of solution-space products is known and has been considered in existing work [17, 34, 47], quantifying this mismatch by computing the actual number of solution-space products for a given software product line is difficult and has not yet been addressed.

In this paper, we solve the problem of computing the actual number of solution-space products by deriving a dedicated solution-space feature model for a software product line. We consider the variability of both, problem and solution space to model the explicit and implicit dependencies between the implementation artifacts. Moreover, we introduce simplifications that improve the overall performance of this derivation. The resulting solution-space feature model allows to apply existing #SAT analyses to compute the exact number of products. The main contributions of our work are:

1. We present a generic concept for deriving solution-space feature models for software product lines. This includes two simplifications which are critical for scalability.
2. We apply #SAT-based analyses to the derived solution-space feature models to quantify the mismatch between problem and solution space.
3. We empirically evaluate the solution-space feature model derivation based on industrial software product lines.

## 2 FEATURE MODELING

The automatic product generation from a software product line [15] is based on three main concepts: *Problem space* (1), *solution space* (2), and *configuration knowledge* (3) [15]. The problem space (1) defines the set of configuration options, that can be selected to create a *configuration*. As not every configuration leads to a valid or useful product, the problem space further defines *dependencies* between the configuration options that limit the valid configurations. The solution space (2) contains the set of *implementation artifacts* (i.e., source code fragments or software modules) that implement the individual configuration options on the technical level. To enable the automatic product generation, the implementation artifacts that are required to build the product for a given configuration need to be identified automatically. Therefore, a software product line further defines the configuration knowledge (3), which is a mapping between configuration options and their required implementation artifacts. There are different variability realization mechanisms to implement this mapping [4, 44]. While in simple cases, a 1:1 mapping between a configuration option and its corresponding implementation artifact can be sufficient, those mappings can be more complex if the selection of an implementation artifact is controlled by multiple configuration options, or if a configuration option is mapped to multiple implementation artifacts.
The configuration knowledge can be formally expressed by presence conditions [4, 14, 16, 29, 31, 33], which are propositional formulas that define for which selection of configuration options a specific solution-space artifact is included in a product. Presence conditions can be retrieved for different variability realization mechanisms that support an automatic product generation [4, 50].

**Definition 2.1** (Presence Condition)**.** Let $i$ be an implementation artifact and let $C_{PS}$ be the set of configuration options in the problem space. The presence condition $p(i)$ of an implementation artifact is a propositional formula over $C_{PS}$ that is satisfied if and only if $i$ is included in the product.

The problem space of a software product line is usually defined by a feature model [8], which represents the configuration options with a set of selectable *features* and defines the dependencies between them with a set of *constraints*. A feature model is commonly visualized by a *feature diagram*, which is a tree structure with a set of nodes that represent available features as shown in Figure 1. This representation also supports the definition of *abstract features* [47], which can be used to structure the feature diagram without representing an actual functionality of the software system. To model the constraints, feature diagrams support different mechanisms. First, the hierarchy is used to express parent-child constraints, where the selection of a child feature requires the selection of its parent feature. Second, those hierarchical constraints can be refined by assigning different modifiers to individual features or groups of features. Marking a feature as *mandatory* states that it needs to be selected if its parent is selected. In contrast, it can be marked as *optional*. Moreover, groups of features can be assigned with one of two modifiers being alternative and or. While an *alternative-group* requires that exactly one feature is selected if its parent feature is selected, an *or-group* requires that at least one feature is selected. Feature diagrams also allow to create *cross-tree constraints* (CTCs) to define dependencies that cannot be modeled by the tree structure itself, such as the dependencies between features in different subtrees. A feature model can be expressed with an equivalent propositional formula [8, 51], which allows to apply external tools such as #SAT solvers [42] which are used in many automatic feature-model analyses [9], such as counting the number of valid configurations.

## 3 RUNNING EXAMPLE

In this section, we introduce a simplified, configurable automotive system which is visualized schematically in Figure 1.
This software product line serves as running example throughout this paper. The problem space defines that the infotainment feature is mandatory for each car configuration. Customers can optionally enable the Bluetooth interface and voice control functionality. Moreover, customers can choose from the two advanced driving assistant systems (ADAS) lane assist and adaptive cruise control (ACC). In

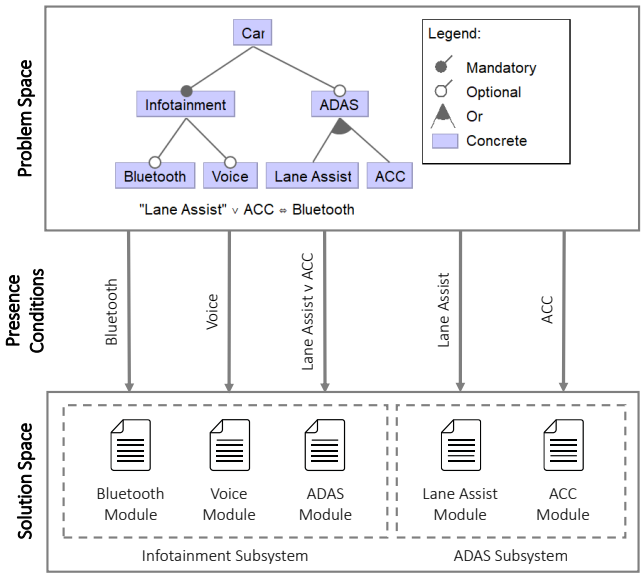


**Figure 1: Simplified Automotive Product Line**

addition, the sales department decided that the Bluetooth interface can only be ordered in combination with at least one ADAS and that the selection of an ADAS also requires the Bluetooth interface. Thus, a cross-tree constraint models this dependency. The solution space contains five implementation artifacts, i.e., software modules, that implement the different functionalities on the software level. The Bluetooth module handles Bluetooth communication, the voice module processes speech input and the ADAS module monitors the status of the assistant systems on the infotainment screen. The lane assist and ACC modules implement the corresponding assistant functionalities. The individual software modules are distributed across two subsystems, one for the infotainment and one for the ADAS. The activation of all five implementation artifacts is controlled by their presence condition. The voice module, for instance, is activated if and only if the voice feature is selected. As the ADAS module in the infotainment subsystem is able to monitor both assistant systems, it is activated if and only if either of both ADAS is selected. In the following sections, we use the infotainment subsystem as example to show why the mismatch between problem-space configurations and solution-space products occurs and how it can be quantified.

## 4 PROBLEM STATEMENT

Highly configurable software product lines hold various challenges such as ensuring that each individual configuration [35] works properly. Thus, the number of possible problem-space configurations is a central indicator for the complexity of a software product line. When reasoning about the software complexity, however, the number of problem-space configurations can be misleading as it does not need to match the number of solution-space products. Computing the number of solution-space products, however, is difficult. This is because there are two aspects that have to be considered when aiming to compute the exact number of solution-space products.

***Implicit Dependencies of Implementation Artifacts.*** The solution-space feature model must consider the implicit dependencies between implementation artifacts that are defined by the problem space. For our running example, such an implicit dependency exists between the Bluetooth module and the ADAS module. Due to their presence conditions in combination with the problem-space constraint $Lane\ Assist \vee ACC \Leftrightarrow Bluetooth$, either both or none of those implementation artifacts are present in a product.

***Problem-Space Collisions.*** Another aspect that needs to be considered are distinct problem-space configurations that lead to the same product. We denote those occurrences as *problem-space collisions* which must be ignored when counting the number of products. For our running example, such problem-space collisions occur when analysing the infotainment subsystem in isolation. As the ADAS module of the infotainment subsystem is enabled if and only if at least one of the assistant systems is selected, there are multiple problem-space configurations that lead to the same infotainment subsystem. Such problem-space collisions, however, do not necessarily indicate a variability bug and can have valid reasons as shown in the running example. In either case, problem-space collisions can result in misleading #SAT analysis results.

## 5 DERIVING A SOLUTION-SPACE FEATURE MODEL

In this section, we show how to derive a dedicated solution-space feature model for a software product line. The concept tackles both aspects that hold potential inaccuracies by considering the implicit dependencies between implementation artifacts and by eliminating problem-space collisions.

To consider the implicit problem-space dependencies, we first create an *integrated feature model* that contains the information of both, problem and solution space. We do this by representing the implementation artifacts with a set of *solution-space features* which we append to the existing problem-space feature model. Further, we append constraints to the integrated feature model that express the logic of the configuration knowledge to connect both spaces.

The integrated feature model, however, is not yet suitable to compute the number of solution-space products, as it is still subject to problem-space collisions. To eliminate them, we apply a slicing algorithm [1] to the integrated feature model which removes all problem-space features while preserving their implicit dependencies between the remaining solution-space features. The resulting solution-space feature model defines the actual set of configurable products of a software product line. In the following, we explain the individual derivation steps in detail based on the infotainment subsystem of our running example.

### 5.1 Implicit Artifact Dependencies

To consider the implicit dependencies between implementation artifacts that result from problem-space constraints, we construct an *integrated feature model* which combines the variability of the problem and solution space. The construction of the integrated feature model is based on the original problem-space feature model. We start with structuring the integrated feature model by inserting two abstract features *Problem Space* and *Solution Space* below the root feature (Figure 2). To represent the solution space, we create a feature for each implementation artifact, which we denote as *solution-space feature* and add them below the solution space sub-tree. We model each solution-space feature as optional feature without any hierarchy. This is because the dependencies of the implementation artifacts result from their presence conditions. The resulting tree structure of the integrated feature model is shown in Figure 2. As the problem space sub-tree remains unchanged, we collapsed it due to space and readability reasons.

Implementation artifacts are typically not freely selectable but bound to their presence condition. Hence, their respective solution-space feature needs to be constrained. For each solution-space feature $i$, we create the constraint

$$i \Leftrightarrow p(i)$$

and append it to the integrated feature model. We call those constraints *solution-space constraints*. The bi-implication is required, as a solution-space constraint needs to cover two directions. On the one hand, a solution-space constraint needs to force the selection of the corresponding solution-space feature if its presence condition is satisfied. On the other hand, the selection of a solution-space feature requires its presence condition to be satisfiable. Thereby, the logic of the solution-space constraints permits to combine solution-space

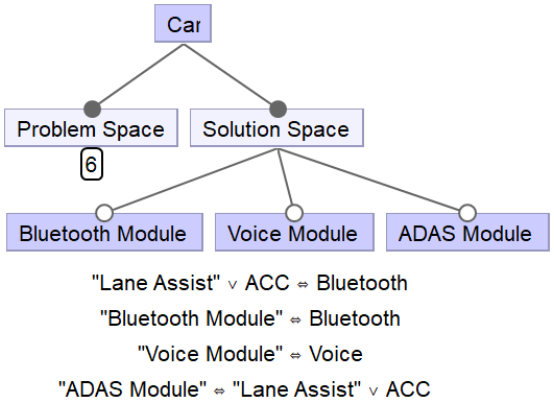


**Figure 2: Integrated Feature Model of the Infotainment Subsystem**

features whose presence conditions are contradictory or to select problem-space features that are incompatible with the presence condition of an already selected solution-space feature. A simple implication (i.e., $i \Rightarrow p(i)$), in contrast, is insufficient, as this would allow to select a solution-space feature even if its presence condition is not satisfied. The complete integrated feature model for the infotainment subsystem of our running example including the solution-space constraints is shown in Figure 2.

By constructing the integrated feature model, the implicit dependencies between implementation artifacts defined in the problem space are captured. Due to the solution-space constraints, the selection of solution-space features only results from problem-space feature selections. Thus, the number of configurations of the integrated feature model is equal to the number of configurations of the original problem-space feature model. Therefore, constructing the integrated feature model is not sufficient to compute the number of solution-space products because problem-space collisions are not yet eliminated.

## 5.2 Eliminating Problem-Space Collisions

To eliminate problem-space collisions, we remove the problem-space features from the integrated feature model while preserving their implicit dependencies between solution-space features. As feature models define the features of a complete system and their dependencies, a crude removal of features may have side effects on the remaining features. To mitigate this, *slicing algorithms* [1, 26, 27] have been developed which allow to remove a set of features from a feature model while preserving all original dependencies between the remaining features.

We consider the constraint $(A \Rightarrow B) \wedge (B \Rightarrow C)$ as an example. When eliminating $B$, the resulting constraint should be equivalent to $(A \Rightarrow C)$, as this preserves the original dependencies between the remaining features $A$ and $C$. The naive removal of variable $B$, however, results in the constraint $(A) \wedge (C)$ which wrongly alters the original dependencies. Figure 3 shows the basic slicing process to obtain the correct constraint for the given example.

The slicing algorithm operates on a conjunctive normal form (CNF), which is why the considered constraint (1) is initially converted into its CNF representation (2). Next, the constraint is duplicated and combined with a logical or (3). The to be eliminated variable $B$ is then replaced with *false* ($\bot$) on the one side and with *true* ($\top$) on the other side (4). The resulting formula is simplified (5) by removing tautology clauses such as $(\neg A \vee \top)$ or unsatisfiable literals such as $\neg\top$. Applying those simplifications leads to the final constraint $\neg A \vee C$ which is equivalent to $A \Rightarrow C$.

(1) **Formula:** $(A \Rightarrow B) \wedge (B \Rightarrow C)$
(2) **CNF:** $(\neg A \vee B) \wedge (\neg B \vee C)$
(3) **Duplicate:** $((\neg A \vee B) \wedge (\neg B \vee C)) \vee ((\neg A \vee B) \wedge (\neg B \vee C))$
(4) **Replace:** $((\neg A \vee \bot) \wedge (\neg\bot \vee C)) \vee ((\neg A \vee \top) \wedge (\neg\top \vee C))$
(5) **Simplify:** $\neg A \vee C$

**Figure 3: The Basic Slicing Process (adapted from [27])**

By slicing the problem-space sub-tree from the integrated feature model, we eliminate the problem-space collisions, but keep the implicit constraints [3] between solution-space features. Thereby, we obtain a feature model that correctly describes the solution-space variability by defining the implementation artifacts and their dependencies, which allows to apply #SAT analysis to compute the number of derivable products. The resulting solution-space feature model for the infotainment subsystem of our running example is shown in Figure 4.

It can be seen that this feature model preserves all dependencies between the solution-space features of the infotainment subsystem. First, all solution-space features remain optional which is correct considering the modeled configurability of the software product line. Moreover, the implicit dependencies between the Bluetooth module and the ADAS module are preserved. As previously explained, due to constraints in the problem-space, either both or none of those artifacts is enabled in a product. This implicit dependency is maintained by the explicit cross-tree constraints (*Bluetooth Module* $\Rightarrow$ *ADAS Module*) and (*ADAS Module* $\Rightarrow$ *Bluetooth Module*). Both constraints result directly from the slicing process and do not require manual overhead.

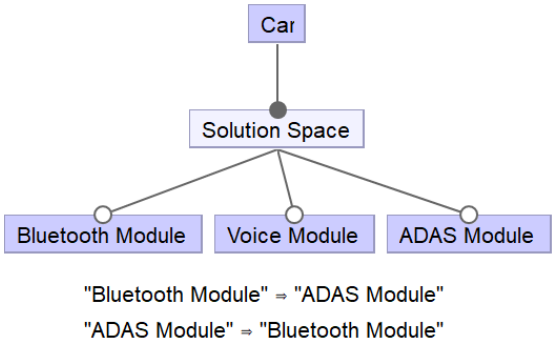


**Figure 4: Solution-Space Feature Model of the Infotainment Subsystem**

## 5.3 Simplifying the Integrated Feature Model

Feature-model slicing is a computationally intensive operation that does not scale well with the size and complexity of a feature model and the number of to be removed features [26]. Depending on the variability realization mechanism and the resulting modularity of the feature implementations, a single feature can be mapped to multiple implementation artifacts. As the integrated feature model represents each implementation artifact with a dedicated solution-space feature along with its solution-space constraint, the integrated feature model can contain high numbers of features and constraints which in turn affects the performance of the slicing algorithm. In preliminary experiments, the derivation of solution-space feature models as described above resulted in long run times ranging from multiple hours up to days for large integrated feature models. Thus, we designed two simplifications that simplify the integrated feature model by (1) eliminating solution-space features and (2) eliminating solution-space constraints while preserving the original solution-space variability and thereby the number of solution-space products. Both simplifications build on *atomic sets* [51].

An atomic set is a set of features that show an equal selection behaviour. In detail, this means that if any feature in the atomic set is (de-)selected, the dependencies defined in the feature model force all other features of the atomic set to be (de-)selected as well. Thus, solution-space features that create an atomic set are always (de-)selected as a unit and can only appear in a product together. Identifying atomic sets in the integrated feature model not only reveals all solution-space features that have the same presence condition (i.e. belong to the same problem-space feature) but also solution-space features that show an equal selection behaviour due to cross-tree constraints.

***Simplification 1: Eliminating Solution-Space Constraints.*** The first simplification step eliminates solution-space constraints by restructuring the integrated feature model based on the atomic set information. The restructuring is performed as follows: For each atomic set of solution-space features, we pick one solution-space feature of that set which we denote as the *representative*. Next, we move the remaining solution-space features of this atomic set below the representative in the feature model and mark them as mandatory. Because those parent-child relations preserve the atomic set relation of involved features, the solution-space constraints of all non-representative solution-space features are no longer required and can be removed. Thus, this operation does not change the number of solution-space products. For each identified atomic set, this operation removes $n - 1$ solution-space constraints from the integrated feature model, where $n$ is the size of the atomic set. Figure 5 shows the optimized integrated feature model for the infotainment subsystem of our running example after applying the elimination of solution-space constraints. As the solution-space features *ADAS Module* and *Bluetooth Module* form an atomic set, they were restructured so that the solution-space feature *Bluetooth Module* becomes a mandatory child of the *ADAS Module*. Hence, the solution-space constraint of the solution-space feature *Bluetooth Module* was removed. Note that both solution-space features could have been picked as representative, as there is no difference in the resulting variability.

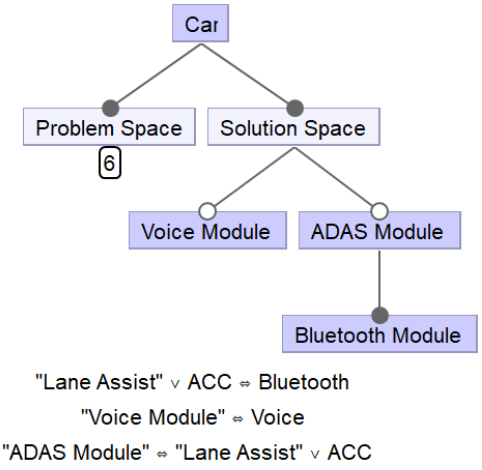


**Figure 5: Integrated Feature Model after Simplification 1**

***Simplification 2: Eliminating Solution-Space Features.*** The second simplification step builds on the first simplification step to further simplify the integrated feature model by removing all non-representative solution-space features of each atomic set. Selection wise, those features behave equally to their previously selected representative solution-space feature and do not introduce any new products. Thus, removing those solution-space features does not affect the number of solution-space products. Because the solution-space constraints of those non-representative solution-space features were already removed during the first simplification, the non-representative solution-space features can be trivially removed from the integrated feature model, as they have no child features and do not appear in any constraint. For our running example, this simplification eliminates the solution-space feature *Bluetooth Module*. Figure 6 shows the further simplified integrated feature model for our running example. When aiming to only compute the number of solution-space products, this simplification can be applied without any limitations. However, this simplification step requires a mapping that indicates, which solution-space feature represents a removed solution-space feature when aiming to reason about individual solution-space features.

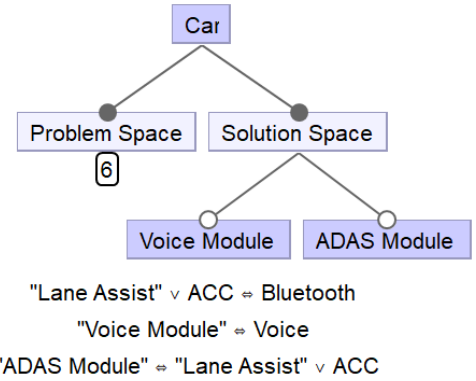


**Figure 6: Integrated Feature Model after Simplification 2**

# 6 #SAT-BASED SOLUTION-SPACE ANALYSES

As we derive a standard solution-space feature model, it allows an out-of-the-box reuse of existing #SAT analyses to analyze the solution space. In this section, we outline four existing #SAT-based problem-space analyses and explain the benefit that is gained when transferring those analyses to the solution space. The analyses are structured into two categories based on their used #SAT query [42].

## 6.1 Cardinality of Feature Models

The first category of #SAT analyses uses the cardinality of a feature model, such as *maintainability prediction* and *variability reduction*

**Definition 6.1** (Feature-Model Cardinality [42])**.** Let $FM$ be a feature model and let $VC$ be the set of satisfying assignments of $FM$. The cardinality of a feature model $\#FM$ is defined as $\#FM = |VC|$.

***Maintainability Prediction.*** Bagheri et al. [6] propose a set of structural metrics to assess the maintainability of a software product line. One of those metrics is to determine the problem-space feature model cardinality of the software product line. The authors argue that a high feature model cardinality decreases the maintainability of a software product line while a low cardinality improves the maintainability. Due to the variability mismatch between problem- and solution space, however, the cardinality of the problem-space feature model can be misleading when reasoning about the maintainability of the technical implementation. For instance, the infotainment subsystem in our running example allows 8 problem-space configurations but only 4 solution-space products, which results in a significant difference in the maintainability metric. Therefore, we argue that measuring the maintainability based on the number of solution-space products increases the reliability of that metric from the software perspective.

***Variability Reduction.*** It is beneficial to reduce the complexity of a software product line to improve its maintainability. This can be achieved by reducing the problem-space variability through feature model adaptions such as adding further constraints. In this scenario, having information about the concrete impact of the adaption onto the software product line is crucial. To determine the impact, we proposed a #SAT analysis to measure variability reduction in earlier work [42].

**Definition 6.2** (Variability Reduction [42])**.** Let $FM$ be a feature model and $FM'$ be an adaption of $FM$. The variability reduction $r$ between the feature models is defined as $r = \#FM - \#FM'$.

When aiming to reduce the software complexity of a software product line, measuring the variability reduction of the problem space is inaccurate. This is because the impact of a feature-model change may differ between problem and solution space. In some cases, the change may even only reduce the number of problem-space configurations while leaving the number of solution-space products unchanged. In our running example, for instance, adding the problem-space constraint $Lane\ Assist \Rightarrow ACC$ only reduces the number of problem-space configurations from 8 to 6 while the number of solution-space products of the infotainment subsystem remains the same. Transferring the analysis to the solution-space feature model eliminates this inaccuracy and provides more detailed information about the actual reduction of the software complexity.

## 6.2 Cardinality of Features

The second category of #SAT analyses is tailored to the cardinality of individual features. The cardinality of a feature is the number of configurations that include that very feature.

**Definition 6.3** (Feature Cardinality [42])**.** Let $FM$ be a feature model and $f$ be a feature. Further, let $VC$ be the set of satisfying assignments of $FM$ and $VC_f \subseteq VC$ be the subset of $VC$ where $f$ is assigned to true. The cardinality of a feature $\#FM_f$ is defined as $\#FM_f = |VC_f|$.

As the feature cardinality is an absolute value, it is often not suitable to compare feature cardinalities across different feature models. Thus, the feature cardinality can be used to compute the relative *commonality* of a feature.

**Definition 6.4** (Feature Commonality [42])**.** Let $FM$ be a feature model and let $\#FM$ be its cardinality. Further, let $f$ be a feature of $FM$ and let $\#FM_f$ be its cardinality. The commonality of a feature is defined as: $Commonality(FM, f) = \begin{cases} 0 & \#FM = 0 \\ \frac{\#FM_f}{\#FM} & \#FM > 0 \end{cases}$

The feature cardinality is used in existing #SAT analyses. We introduce two of those analyses for which we argue that they provide more detailed information when applied to the solution space.

***Degree of Reuse.*** Cohen et al. [13] proposed to use the feature cardinality to quantify the degree of reuse for a feature. This metric is used to rate the benefit of the software product line approach instead of separate software products. Features with a high cardinality are assigned with a high degree of reuse because they can be selected in many different configurations. As features with a low cardinality can only be selected in few configurations, such features are assigned with a low degree of reuse.
In a software product line, the elements that are actually reused are the implementation artifacts. Thus, applying this analysis to the problem-space features provides inaccurate results. Being able to determine the degree of reuse for specific solution-space features eliminates this inaccuracy. For instance, the presence condition of the *ADAS Module* is satisfied in 6 out of 8 (75 %) configurations. The cardinality of the *ADAS Module* itself, however, is 2 (50 %) resulting in a significant difference in the degree of reuse metric.

***Feature Prioritization.*** Another analysis that is based on feature cardinality is the derivation of a feature prioritization that we proposed in earlier work [42]. The authors present two different use cases. First, the prioritization can be used to decide which feature is to be implemented first. Here, the authors argue that features with a high cardinality are to be prioritized as this allows to create more distinct configurations. Second, the prioritization can be used during testing. Sundermann et al. [42] argue that features with a low cardinality should be prioritized during testing as those features are more easy to miss while features with a high cardinality are more likely to be tested anyways.
Implementing and testing the source code of a software product line are solution-space related processes. Thus, the feature prioritization can be refined when considering the cardinality of solution-space features instead of problem-space features as this mitigates the inaccuracies that stem from the mismatch between problem-space configurations and solution-space products.

# 7 EVALUATION

We evaluate solution-space feature model derivation and the #SAT-based solution-space analyses empirically by applying them to industrial software product lines. Based on the evaluation results, we aim to answer the following research questions:

***RQ1:*** *How does solution-space feature model derivation scale for industrial software product lines regarding run time?*
The derivation of the solution-space feature models includes multiple complex steps. Especially the slicing step is computationally intensive [26] and may result in long run times when applied to large integrated feature models. Hence, we evaluate the run time of solution-space feature model derivation when applied to industrial software product lines. Capturing the impact of the simplifications that we introduced in Section 5.3 is a central aspect of this evaluation. To answer this research question, we derive the solution-space feature model for the complete subject systems and all of their subsystems and capture the required run times. To evaluate the impact of the individual simplifications that we introduced in Section 5.3, we perform the derivation in three scenarios. While we apply no simplifications in the scenario *None*, we apply the elimination of solution-space constraints in scenario *SSCs*. In scenario *SSCs+SSFs*, we additionally apply the elimination of solution-space features.

***RQ2:*** *How do feature and feature-model cardinalities differ between problem and solution space for industrial software product lines?*
In this paper, we argue that the number of solution-space products can differ from the number of problem-space configurations due to complex mappings between problem-space features and implementation artifacts. However, the actual difference between both numbers is not known for industrial software product lines. Thus, we examine this difference by computing the cardinalities of the problem and solution-space feature models of industrial software product lines. To answer this research question, we first compute the partial problem-space feature models of the subsystems by slicing all problem-space features that do not affect the respective subsystem (i.e., features that do not appear in any presence condition of the subsystem's implementation artifacts) from the problem-space feature model. The partial problem-space feature models serve as baseline for the comparison. We then derive the solution-space feature model for each subsystem and compare the cardinality of their partial problem-space feature model and solution-space feature model. For subsystems that are not affected by a mismatch, both cardinalities are expected to be equal.

## 7.1 Experimental Setup

We evaluate solution-space feature model derivation based on three subject systems that were provided by our industry partner. We selected those subject systems because it allows to discuss the input data as well as the results with domain experts to verify our results and prevent potential errors. We perform the evaluation by deriving the solution-space feature models for those subject systems and use #SAT analyses to quantify the proposed mismatch between problem-space configurations and solution-space products. In the following section, we outline the structure of the input data which we used to derive the solution-space feature models.

***Problem-Space Feature Model.*** The problem-space variability of the subject systems was not initially modeled by a feature model. However, the configuration options and their constraints were given in a formal, machine readable format which allowed a direct translation of the given input data into a feature model. The translation was performed in close exchange with experts to mitigate translation errors.

***Implementation Artifacts and Presence Conditions.*** For each of the three subject systems, the set of implementation artifacts and their individual presence conditions were explicitly known. Moreover, the distribution of implementation artifacts across specific subsystems, called electronic control units (ECUs) in the automotive context, was given. The granularity of implementation artifacts ranged from software modules to individual parameters. As this data was provided to us in a machine readable format, we were able to directly process this data without the need for preprocessing. We discussed the format and the semantics of the provided data with the corresponding data experts beforehand to mitigate errors during the construction of the integrated feature models. Table 1 shows the solution-space structure of the subject systems. It shows the number of subsystems per subject system, their size regarding implementation artifacts and the total number of implementation artifacts of the subject systems.

| Subject System | #Subsystems | #Impl. Artifacts | #Total Impl. Artifacts |
|---|---|---|---|
| Automotive System A | 33 | 3 - 1216 | 4900 |
| Automotive System B | 42 | 2 - 6194 | 10717 |
| Automotive System C | 44 | 2 - 5939 | 13643 |

**Table 1: Subject Systems and their Solution-Space Structure**

***Implementation.*** We implemented the algorithm for deriving a solution-space feature model based on the FeatureIDE library v3.7.2[1] [32]. This library not only supports the creation of feature models but also provides efficient algorithms for feature model slicing [27] and the identification of atomic sets. As the FeatureIDE library has no build-in #SAT support, we use an external #SAT-solver[2] to perform the feature-model counting. The evaluation was performed on a conventional notebook with following specifications: Intel i7-9850H @2,60 GHz, 32 GB RAM, Windows 10 64 Bit. As the implementation is subject to closed source restrictions, we cannot provide it in a replication package.

## 7.2 Results

In this section, we present the results that we obtained by capturing the run times of the solution-space feature model derivation for the subject systems described above. Moreover, we show the results of the #SAT queries that we applied to the problem-space feature models and the derived solution-space feature models.

[1] https://featureide.github.io/
[2] https://sites.google.com/site/marcthurley/sharpsat

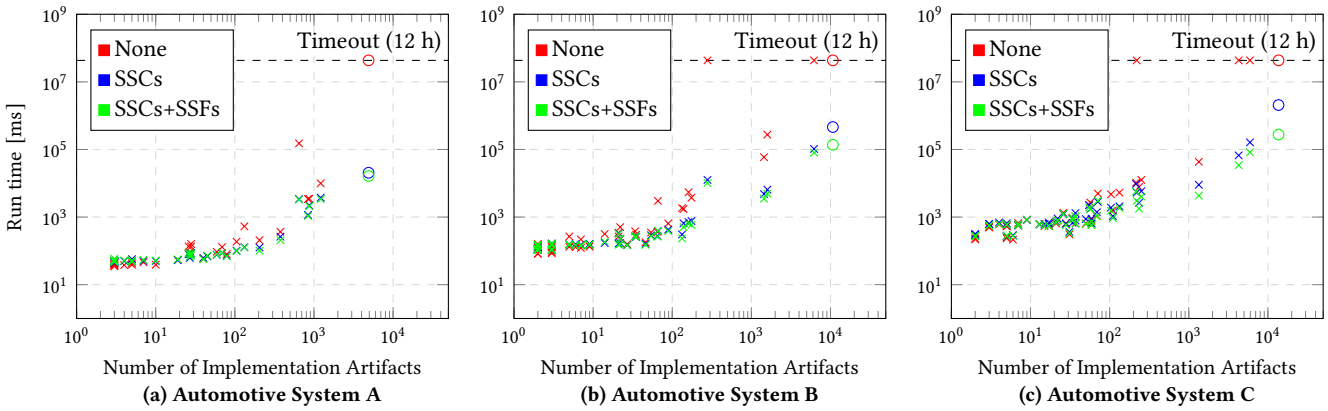


**Figure 7: Impact of the Simplifications onto the Total Derivation Run Time per Subject System**

***Run Time Behaviour.*** To capture the run time behaviour of our algorithm, we derived the solution-space feature model for each complete subject system as well as for their individual subsystems. To determine the impact of the simplifications onto the run time, we performed the solution-space feature model derivation for all three simplification scenarios. To increase the reliability of the captured data, we computed the average run time of each derivation over 10 runs. The obtained results for the three subject systems are visualized in separate charts in Figure 7. The charts show the total run time required for deriving the solution-space feature model in relation to the total number of implementation artifacts. The different simplification scenarios are visualized in separate series. Data points marked with a cross indicate the run times for individual subsystems, while a circle indicate the run time for the complete subject systems.

The results show that the run time of the solution-space feature model derivation increases with the number of implementation artifacts. However, it can also be seen that the number of implementation artifacts is not the only factor that affects the run time, as there are subsystems with fewer implementation artifacts, for which the derivation took longer compared to larger subsystems of the same subject system. Moreover, the required run time for subsystems with a similar number of implementation artifacts shows immense differences across the three subject systems. For instance, the solution-space feature model derivation for a subsystem of *Automotive System B* with about **40** implementation artifacts took about **10,000 ms**. This is about **50** times longer compared to a subsystem with a similar number of implementation artifacts in *Automotive System A*, which took about **200 ms**. In general, the results not only confirm a substantial run time reduction when using our simplifications but also show that the simplifications are crucial for scalability, as the solution-space feature models for the complete subject systems and some large subsystems were only derivable with enabled simplifications. For the individual subsystems, the average run time reduction between the simplification scenarios *None* and *SSCs+SSFs* amounts to **41.9 %** with a maximum reduction of **99.99 %** and a minimum reduction of **0.009 %**. For the complete systems, the average run time reduction amounts to **99.7 %** with a maximum reduction of **99.9 %** and a minimum reduction of **99.4 %**.

***Cardinalities of Solution-Space Feature Models.*** To quantify the mismatch between the problem-space configurations and solution-space products, we computed the cardinalities of the (partial) problem-space feature models of all (sub-) systems and their derived solution-space feature models. The chart in Figure 8 visualizes the results of this analysis. It shows the cardinality of the problem-space feature model related to the cardinality of the solution-space feature model. The data points that belong to subsystems are marked with a cross while the data points that belong to complete systems are marked with a circle. The results show that the number of solution-space products is always lower or equal to the number of problem-space configurations. In detail, **84 (70.5 %)** of all 119 subsystems show a deviation between the number of solution-space products and problem-space configurations and are therefore affected by the mismatch. Moreover, it can be seen that also all complete systems are affected by the mismatch. Only **35 (29.5 %)** smaller subsystems with $\leq$ **11** problem-space configurations are not affected by the mismatch. The results further show that the degree of the mismatch varies greatly. For the individual subsystems, the average mismatch is **44.7 %** with a maximum mismatch of **99.7 %** and a minimum mismatch of **19.9 %**. For the complete systems, the average mismatch amounts to **97.7 %** with a maximum mismatch of **98.9 %** and a minimum mismatch of **96.1 %**. The degree of the mismatch, however, does not correlate with the total number of problem-space configurations or the number of solution-space products.

***Cardinalities of Solution-Space Features.*** We examined how the variability mismatch between problem and solution space affects the individual implementation artifacts. To do this, we computed the commonalities of solution-space features in the integrated feature-model and compared them to to their commonalities in the solution-space feature model. Figure 9 visualizes the results of this analysis. The chart shows the commonality of each representative solution-space feature in the integrated feature model related to the commonality of that feature in the solution-space feature model. The data shows that the commonalities do not match for the majority of solution-space features. In contrast to the feature-model cardinalities, however, the solution-space feature commonalities

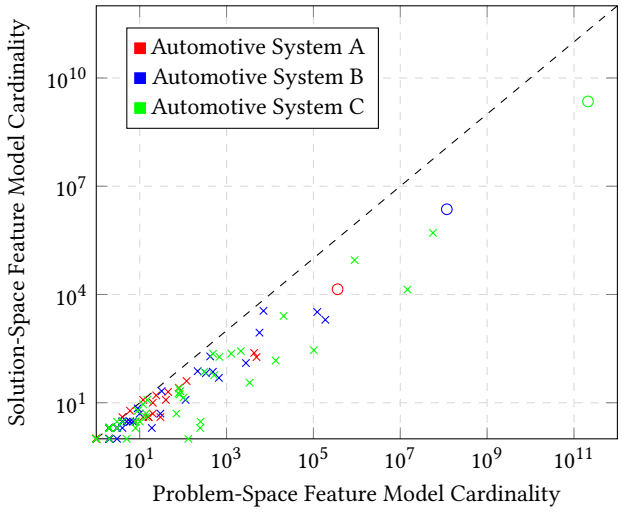


**Figure 8: Cardinality Mismatch Between Problem and Solution-Space Feature Model for each Subject System**

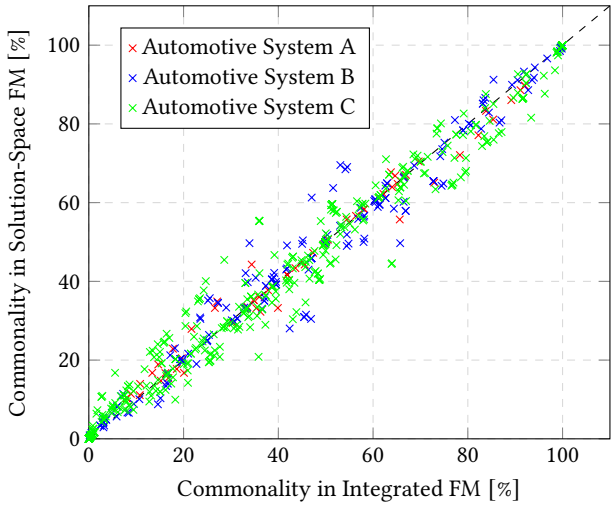


**Figure 9: Commonality Mismatch of Solution-Space Features Between Integrated- and Solution-Space Feature Model**

are subject to a positive and negative mismatch. The data shows that the mismatch applies to **96.6 %** of all solution-space features with an average absolute mismatch of **3.1 %**, a maximum absolute mismatch of **19.47 %** and a minimum absolute mismatch of **0.01 %**.

## 7.3 Discussion

In this section, we discuss the results of our evaluation. We conclude our findings and answer the previously defined research questions.

***RQ1:*** *How does solution-space feature model derivation scale for industrial software product lines regarding run time?*
The evaluation has shown that applying the proposed simplifications are crucial for the run time scalability of the solution-space feature model derivation. Without the simplifications, deriving the solution-space feature model for the complete systems and for larger subsystems ran into the defined timeout of 12 hours. Applying the simplifications prevented the timeout in all cases and even reduced the derivation run time for smaller subsystems. Only for the smallest subsystems ($\leq$ 11 implementation artifacts) the computational overhead of the simplifications resulted in longer run times. We further found that the elimination of solution-space features has only little impact onto the run time for smaller subsystems, but results in an immense run time reduction for larger subsystems and the complete subject systems.

***RQ2:*** *How do feature and feature-model cardinalities differ between problem and solution space for industrial software product lines?*
The #SAT analyses revealed that the majority of the analyzed subsystems as well as the complete systems show differing feature and feature-model cardinalities between the problem and solution-space, making them affected by the variability mismatch. Our results show that this mismatch can be substantial with a maximum of 99.7 % less products than configurations for the individual subsystems and a maximum of 98.9 % less products than configurations for the complete systems. Further, we have shown that this mismatch also affects the cardinality of individual features. On this level, we have found that the mismatch can not only be negative (an implementation artifact occurs in less products than configurations) but also positive (an implementation artifact occurs in more products than configurations). The latter results from the mismatch of the complete (sub)-system itself.

## 7.4 Threats to Validity

Due to the complexity of the processed data and the used mechanisms, the obtained evaluation results are subject to different threats to validity. In this section, we point out the individual threats and explain how we mitigated them.

***Internal Validity.*** There are three aspects that potentially affect the internal validity of the results obtained in our evaluation. The first and most central aspect is the correct translation of the input data into the integrated feature model. To prevent translation errors, we discussed the input data as well as the translation in detail with domain experts. The second aspect is the correctness of the algorithms that we used for the atomic set identification and the feature-model slicing. To mitigate this threat, we chose to use the atomic set and slicing algorithms implemented in the FeatureIDE library, as this is a common and frequently used tool not only in research but also in industry [2, 21]. The last aspect that we considered is the correctness of the simplification implementation, which we ensured by verifying that the simplifications do not alter the variability of the solution-space feature model. This was done by checking the cardinalities of all solution-space feature models that we obtained with different simplification scenarios. In every case, the cardinalities matched for each simplification scenario.

***External Validity.*** Although our presented approach for deriving a solution-space feature model is generic, we cannot state it's general applicability for arbitrary software product lines. This is

because we could only evaluate it on proprietary software product lines, due to the required data and subsystem information that are not present for publicly available product lines. Moreover, our evaluation has shown that the run time of the derivation increases with the number of implementation artifacts. Thus, we cannot state that the performance of current state-of-the-art slicing algorithms is sufficient to execute the solution-space feature model derivation for arbitrary software product lines. Finally, the mismatch that we could confirm for the evaluated automotive subject systems cannot be assumed to be representative for other software product lines.

## 8 RELATED WORK

Our concept for deriving a solution-space feature model is based on existing mechanisms and is related to existing work, especially in the context of #SAT applications and product line analyses.

***Feature-Model Counting.*** Heradio et al. [22, 23], Fernandez-Amoros et al. [18], Sundermann et al. [42, 43], Bagheri et al. [7], Clements et al. [12], and Benavides et al. [9] present feature-model analyses dependent on counting. However, each proposed application only considers the problem space while our analyses provide insights on the solution space. Moreover, several approaches for computing the number of valid configurations have been proposed. Pohl et al. [36] compare the run times of different types of solvers for feature-model analyses, including counting. Kübler et al. [28] employ #SAT solvers on multiple versions of an automotive product line. Sundermann et al. [43] evaluate a variety of #SAT solvers on industrial feature models. Fernandez-Amoros et al. [18] propose an algorithm that separates feature tree and cross-tree constraints to compute the number of valid configurations. Each of the described approaches is limited to the problem space. Nevertheless, using our concept would allow to apply each of the listed approaches also to the solution space.

***Mismatch Between Problem and Solution Space.*** The mismatch between the problem and solution space is known but has not been quantified so far. El-Sharkawy et al. [17] analyzed the Linux Kernel and introduced different types of mismatches between configurations and implementation artifacts. Thüm et al. [47] have shown the mismatch between problem-space configurations and solution-space products by identifying features that are not mapped to an implementation artifact. However, we have shown that this is not sufficient to eliminate the mismatch completely as shown in Figure 8. Moreover, Nadi et al. [34] reverse engineered a problem-space feature model from solution-space implementation artifacts with the scope of ensuring only well-typed configurations. Although this approach helps to reveal constraints that are required to avoid type errors, it does not allow to compute the number of solution-space products as it is limited to the problem space.

***Feature-Model Slicing.*** Feature-model slicing has only been applied to the problem space. Acher et al. [1] and Thüm et al. [47] propose to use slicing on feature models and introduce different applications. Krieter et al. [27] present an efficient slicing algorithm that is based on logical resolution instead of the existential quantification of propositional variables. This algorithm is implemented in the FeatureIDE library and was used to implement our solution-space feature model derivation. Ananieva et al. [3] use slicing algorithms for computing domain specific problem-space views from a common feature model to reveal implicit constraints between the domains. In other work, slicing algorithms have been used to compute feature-model interfaces to speed up feature-model analyses [40] and the product configuration [30]. Another slicing application is introduced by Thüm et al. [48] who derive partial problem-space feature models to simplify the formal verification of configurable systems.

***Analyses of Software Product Lines.*** Besides software product line analyses that are limited to the problem space [9], there are multiple analyses that also consider actual implementation artifacts [46]. Gazzillo et al. [20] and Kästner et al. [25] both present approaches to verify the syntactical correctness of the variable source code for all configurations, by statically checking all possible interactions between implementation artifacts. To reveal type errors that stem from variability, Aversano et al. [5] present an approach to detect incompatibilities between all potential types of variable and its usages. A similar approach is presented by Kästner et al. [24] who use a product-line-aware type system to type check a software product line without the need of generating specific products. To identify run time and behavioral errors of software product lines, Post and Sinz [37] and Classen et al. [11] reuse existing verification techniques such as model checking and deduction-based approaches to verify all configurations. Liebig et al. [29] further propose a variability-aware liveness analysis suitable to analyze the source code of a software product line. Besides those errors, Tartler et al. [45] have proposed an analysis that considers the mapping between problem and solution-space to find dead and superfluous code blocks. Moreover, Gazillo et al. [19] retrieve the mapping with a static source code analysis. Von Rhein et al. [50] present a technique to ease the manual reasoning about such mappings by simplifying the respective presence conditions.

## 9 CONCLUSION AND FUTURE WORK

In this paper, we solved the problem of computing the number of solution-space products by deriving a solution-space feature model that allows to reuse existing #SAT analyses. Our evaluation shows that the run time of the solution-space feature model derivation scales for industrial feature models when applying our proposed simplifications. With the results of the #SAT analyses, we confirmed and quantified the mismatch between problem-space configurations and solution-space products for the evaluated automotive systems. With a maximum mismatch of 98.9 %, the results show that it can be highly inaccurate to reason about the software complexity of a software product line based on the number of problem-space configurations. We argue that a solution-space feature model opens new applications for existing feature-model analyses such as detecting dead code (i.e., dead solution-space features) with a dead feature analysis. Another promising application are feature-model sampling algorithms [49]. In future work, we aim to use solution-space feature models for sampling directly on the solution space to derive more accurate samples.

## 10 DISCLAIMER

The results, opinions and conclusions expressed in this work are not necessarily those of Volkswagen Aktiengesellschaft.

# REFERENCES


[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 424–427.

[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 144–155.

[3] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 18–27.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[5] Lerina Aversano, Massimiliano Di Penta, and Ira D. Baxter. 2002. Handling Preprocessor-Conditioned Declarations. In *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. IEEE, 83–92.

[6] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal (SQJ)* 19, 3 (2011), 579–612.

[7] Ebrahim Bagheri, Tommaso Di Noia, Dragan Gasevic, and Azzurra Ragone. 2012. Formalizing Interactive Staged Feature Model Configuration. *J. Software: Evolution and Process* 24, 4 (2012), 375–400.

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

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

[10] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. on Software Engineering (TSE)* 39, 8 (2013), 1069–1089.

[11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic Model Checking of Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 321–330.

[12] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report. Carnegie-Mellon University.

[13] Sholom Cohen. 2003. *Predicting When Product Line Investment Pays*. Technical Report. Carnegie-Mellon University.

[14] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. Springer, 422–437.

[15] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.

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

[17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 19–28.

[18] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. 2014. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *IEEE Trans. on Software Engineering (TSE)* 40, 9 (2014), 895–910.

[19] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 279–290.

[20] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 323–334.

[21] Marc Hentze, Tobias Pett, Thomas Thüm, and Ina Schaefer. 2021. Hyper Explanations for Feature-Model Defect Analysis. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 14, 9 pages.

[22] Ruben Heradio, David Fernández-Amorós, José Antonio Cerrada, and Ismael Abad. 2013. A Literature Review on Feature Diagram Product Counting and Its Usage in Software Product Line Economic Models. *Int'l J. Software Engineering and Knowledge Engineering (IJSEKE)* 23, 08 (2013), 1177–1204.

[23] Rubén Heradio-Gil, David Fernández-Amorós, José Antonio Cerrada, and Carlos Cerrada. 2011. Supporting Commonality-Based Analysis of Software Product Lines. *IET Software* 5, 6 (2011), 496–509.

[24] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-Based Product Lines. *Trans. on Software Engineering and Methodology (TOSEM)* 21, 3 (2012), 14:1–14:39.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 805–824.

[26] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 60–64.

[27] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. 2016. *An Efficient Algorithm for Feature-Model Slicing*. Technical Report FIN-001-2016. University of Magdeburg.

[28] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 44–53.

[29] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

[30] Michael Lienhardt, Ferruccio Damiani, Einer Broch Johnsen, and Jacopo Mauro. 2020. Lazy Product Discovery in Huge Configuration Spaces. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 1509–1521.

[31] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management With the Virtual Platform. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 1658–1670.

[32] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[33] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Article 14, 11 pages.

[34] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. on Software Engineering (TSE)* 41, 8 (2015), 820–841.

[35] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 78–83.

[36] Richard Pohl, Kim Lauenroth, and Klaus Pohl. 2011. A Performance Comparison of Contemporary Algorithmic Approaches for Automated Analysis Operations on Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 313–322.

[37] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 347–350.

[38] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *Software and System Modeling (SoSyM)* 16, 1 (2017), 77–96.

[39] Hamideh Sabouri and Ramtin Khosravi. 2010. An Effective Approach for Verifying Product Lines in Presence of Variability Models. In *Proc. Int'l Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, Goetz Botterweck, Stan Jarzabek, Tomoji Kishi, Jaejoon Lee, and Steve Livengood (Eds.). Lancaster University, 113–120.

[40] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678.

[41] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 33–42.

[42] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 12, 10 pages.

[43] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 3, 9 pages.

[44] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A Taxonomy of Variability Realization Techniques: Research Articles. *Software: Practice and Experience* 35, 8 (2005), 705–754.

[45] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 81–86.

[46] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45.

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

[48] Thomas Thüm, Tim Winkelmann, Reimar Schröter, Martin Hentschel, and Stefan Krüger. 2016. Variability Hiding in Contracts for Dependent Software Product

Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 97–104.

[49] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 1–13.

[50] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 178–188.

[51] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. In *Proc. Int'l Conf. on Formal Engineering Methods (ICFEM)*. Springer, 115–130.