Generic Solution-Space Sampling for Multi-domain Product Lines

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

Tobias Pett
t.pett@tu-braunschweig.de
Technical University Braunschweig
Germany

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

Sebastian Krieter
sebastian.krieter@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

Validating a configurable software system is challenging, as there are potentially millions of configurations, which makes testing each configuration individually infeasible. Thus, existing sampling algorithms allow to compute a representative subset of configurations, called sample, that can be tested instead. However, sampling on the set of configurations may miss potential error sources on implementation level. In this paper, we present solution-space sampling, a concept that mitigates this problem by allowing to sample directly on the implementation level. We apply solution-space sampling to six real-word, automotive product lines and show that it produces up to 56 % smaller samples, while also covering all potential error sources missed by problem-space sampling.

CCS Concepts: • **Software and its engineering → Software product lines**; **Software testing and debugging**; **Feature interaction**.

Keywords: Multi-domain product lines, feature-model sampling, solution-space analysis

ACM Reference Format:
Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 2022. Generic Solution-Space Sampling for Multi-domain Product Lines. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564719.3568695

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568695

1 Introduction

Today, software-intensive systems are often highly configurable to meet the individual needs of customers. Such systems can be developed as a product line [10], which is a platform that defines a set of abstract configuration options and maps them to a corresponding set of concrete implementation artifacts. This mechanism allows to automatically generate individual products by assembling all required implementation artifacts for a given configuration, a set of selected configuration options. The set of valid configurations is formally referred to as problem space, while the set of derivable products is denoted as solution space.

Validating and testing a product line is challenging, as the problem space of a product line is often too large to test each valid configuration individually [13, 22]. To mitigate this, there are different approaches to compute a representative subset of configurations, called sample, that can be tested instead [1, 17, 21, 26]. One approach to compute such a sample is combinatorial interaction sampling, which generates a set of configurations that covers certain interactions between configuration options.

However, a problem-space sample is not necessarily also suitable to validate the set of derivable products, i.e., the solution space. This is because of two aspects. First, the problem-space sample may consider distinct interactions of configuration options that lead to the same interaction of implementation artifacts. Such redundancies impair the quality of the sample due to an unnecessary large number of configurations. Second, the problem-space sample may miss implementation artifacts or interactions between them. This also impairs the quality of the sample, as some implementation artifacts or interactions that could cause an error have never been included in a product, and thus, have not been tested.

Sampling on a specific domain of a multi-domain product line, e.g., to validate the software domain in isolation, is especially prone to redundant and missed solution-space interactions. However, the size of the generated sample is highly important, as testing a particular configuration of a multi-domain product line, e.g., by equipping a software

in-the-loop [4, 7] test rack, is often costly. Moreover, the coverage of all implementation artifacts and their interactions is crucial, as untested artifacts or interactions between them may result in undetected faults.

In this paper, we present our concept for *solution-space sampling* that reaches *solution-space coverage* by ensuring that all relevant interactions between implementation artifacts are covered, while redundant interactions are omitted. Our concept is based on integrated feature models that we introduced in previous work [16]. We adapt integrated feature models for multi-domain product lines and use them for solution-space sampling by applying existing feature-model sampling algorithms. Our concept also supports to focus the sampling on specific parts of the solution space, e.g., to validate the software domain in isolation. However, it may also be necessary to test the sampled domain in the context of a complete product including all domains. For instance, the software domain should also be tested in a real car to establish the functional correctness of the software in a realistic environment. As there can be multiple matching car configurations for a given software configuration, selecting one of the possible car configurations becomes an optimization problem. Thus, our concept includes an optimized configuration completion based on integer linear programming, which allows to, e.g., find the cheapest complete car for a given software configuration. The main contributions of our work are:

- We introduce a generic approach for solution-space sampling on multi-domain product lines.
- We use integer linear programming to compute optimized complete configurations for domain-specific solution-space samples.
- We evaluate solution-space sampling and the integer linear programming optimization on industrial, automotive product lines.

2 Background and Running Example

The generic solution-space sampling presented in this paper is based on multi-domain product lines, feature models, feature-model sampling and integer linear programming. In this section, we explain those concepts based on a simplified automotive product line, which also serves as running example throughout this paper.

2.1 Multi-Domain Product Lines

A product line is based on three principles that allow an automatic product generation. First, a product line defines available configuration options [10]. A concrete selection of those configuration options is called configuration. Further, a product line defines dependencies between configuration options to prevent invalid or undesired configurations. The resulting set of valid configurations is referred to as problem space [10]. Second, a product line defines a set of implementation artifacts (e.g., source code fragments or software modules) that implement the configuration options on the technical level. Third, a product line defines the configuration knowledge [10] which maps each configuration option to its required implementation artifacts. The configuration knowledge can be formally expressed by presence conditions [3, 9, 11, 23, 24, 28], which are propositional formulas that define under which selection of configuration options a specific implementation artifact is included in a product. Presence conditions can be derived for different variability mechanisms, including preprocessor directives [18, 34]. Based on this product-line structure, a configuration can be used to automatically identify the required subset of implementation artifacts to assemble an individual product. The resulting set of derivable products is referred to as solution space [10].

To define the problem space of a product line, feature models [6] are frequently used. A feature model represents available configuration options with a set of features and defines the dependencies between them with a set of constraints. A feature model can be visualized by a feature diagram, which is a tree-structural representation of the features. Abstract features [33] can be used to structure the feature-model tree and do not represent any actual functionality. To specify the constraints between features, there are different mechanisms. First, the hierarchy of features in the tree structure is used to model parent-child relations, where the selection of a child feature requires the selection of its parent feature. Feature models further allow to assign features with different modifiers. A feature marked as mandatory needs to be selected if its parent is selected, while an optional feature does not need to be selected. Moreover, a set of features can be assigned with different group types. An or-group requires to select at least one of the contained features if the parent feature of the group is selected, while an alternative-group requires that exactly one feature of the group is selected. The third mechanism to model dependencies between features is to define explicit constraints in propositional logic, called cross-tree constraints, which are used to express dependencies that cannot be modeled by the tree structure, such as dependencies between features in different sub-trees.

In a multi-domain product line, the derivable products are assembled from implementation artifacts of different domains such as software, electronics and mechanics. In this context, we define the domain-relevant problem and solution space for a domain d.

Definition 2.1 (Domain-Relevant Problem Space)**.** Let F be the set of problem-space features. The domain-relevant problem space PS_d of d is the set of configurations that can be created from the subset of problem-space features $F_d \subseteq F$ that affect d (i.e., appear in a presence condition of at least one implementation artifact of d). We call the configurations in PS_d domain configurations of d.

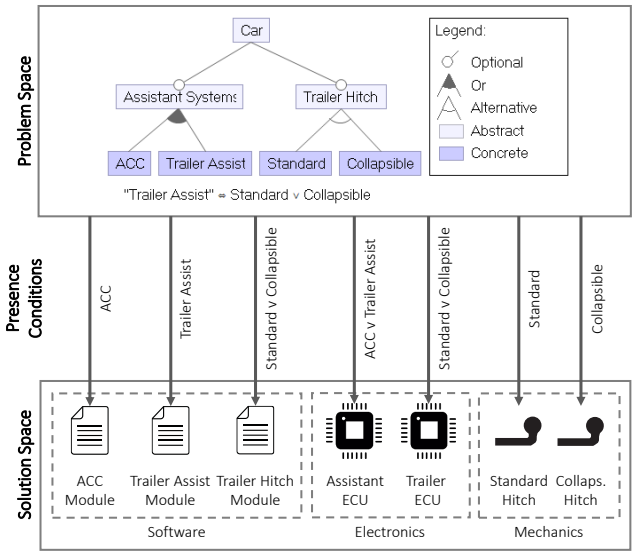

Figure 1. Simplified Multi-Domain Product Line

Definition 2.2 (Domain-Relevant Solution Space)**.** Let A be the set of implementation artifacts. The domain-relevant solution space SS_d of d is the set of products constructable from the subset of implementation artifacts $A_d \subset A$ that belong to d. We call a product in SS_d partial product of d.

The structure of a multi-domain product line is schematically visualized in our running example in Figure 1. This simplified, automotive multi-domain product line allows to configure a car via four configuration options, whose dependencies are defined by a feature model. The customer can choose from two assistant systems, adaptive cruise control (ACC) and a trailer assist. Moreover, the customer can choose between a standard trailer hitch and a collapsible trailer hitch. Each configuration option is mapped to implementation artifacts of different domains. There are three software modules that implement the functionalities on the software level, two electronic control units (ECUs) on which the software is deployed, and the two different mechanical trailer hitches.

2.2 Feature-Model Sampling

Testing a product line is challenging, as it theoretically requires to test each valid configuration. However, due to combinatorial explosion, a product line often allows to create millions of valid configurations, which makes testing each configuration individually infeasible. Thus, there are different approaches to derive a representative subset of configurations, called sample, that can be tested instead. One of those approaches is combinatorial interaction testing, which is based on the assumption that errors in a particular configuration result from interactions between features. A feature interaction is a combination of feature selections. For instance, the two features *Standard* and *Collapsible* from our running example have a total of four possible interactions (¬ indicates the deselection of a feature):

1. (¬*Standard*, ¬*Collapsible*)
2. (¬*Standard*, *Collapsible*)
3. (*Standard*, ¬*Collapsible*)
4. (*Standard*, *Collapsible*)

T-wise interaction-sampling algorithms, such as YASA [21], IncLing [1] and ICPL [17] build onto this concept. Those algorithms generate a set of configurations, where each interaction between t features is contained in at least one configuration. For instance, sampling with t = 2 means that all interactions between all *pairs* of features need to be covered by the configurations contained in the sample. Interactions that are invalid, i.e., interactions that cannot occur in a configuration due to constraints in the feature model, are not considered. In our running example, the interaction *{Standard, Collapsible}* is invalid, because the feature-model constraints resulting from the alternative group forbid that both features are selected together in a configuration.
The quality of a sample results from two properties. First, the *efficiency* of a sample indicates the effort that is required to validate a system based on the sample. This property is commonly determined by the size of the sample and the time that is required to compute the sample. Second, the *effectiveness* of a sample indicates how suitable a sample is to find errors in the target system, which is commonly determined by how representative the sample is. In the case of t-wise interactions sampling, this property is affected by the selected t and the share of interactions that are covered.

2.3 Integer Linear Programming (ILP)

An integer linear program is an approach to solve optimization problems, where variables have to be assigned with specific integer values to maximize or minimize a given objective function. To limit the possible variable assignments, integer linear programs support the definition of linear constraints between variable values. Further, variables can be multiplied by a coefficient, e.g., to encode their weighting in the objective function. In the context of feature models, integer linear programming can be used to compute optimized configurations, e.g., to find the cheapest valid configuration. To apply ILP, the feature model needs to be translated into an integer linear program that preserves features and their dependencies. To perform this translation, the feature model first needs to be translated into an equivalent propositional formula [25] that is commonly given in conjunctive normal form (CNF). A propositional formula in CNF is a conjunction of clauses, where each clause is a disjunction of literals. A literal, in turn, is a boolean variable or its negation. The structure of such formula allows a generic translation into an ILP.

Encoding Features. First, the features of the feature model are encoded into ILP variables. As a feature is a boolean variable, i.e., selected or deselected, its respective ILP variable needs to be constrained to only take on the values zero (deselected) or one (selected). Thus, for each feature f_i, we define an ILP variable x_i and the ILP constraint

$$0 \leq x_i \leq 1 \mid x_i \in \{0, 1\} \tag{1}$$

Encoding a Logical Not. The logical not requires that its respective literal is deselected. This boolean constraint can be encoded by a helper variable y_i and the ILP constraint

$$y_i = 1 - x_i \tag{2}$$

Encoding a Logical Or. The logical or operator requires that at least one contained literal evaluates to true. Thus, a logical or with n literals $x_1, ..., x_n$ can be encoded by a helper variable z_i and the ILP constraints

$$z_i \leq x_1 + x_2 + ... + x_n \tag{3}$$

$$z_i \geq x_1, z_i \geq x_2, ..., z_i \geq x_n \tag{4}$$

$$0 \leq z_i \leq 1 \tag{5}$$

Encoding a Logical And. An explicit encoding of the logical And is not required. This is because the defined constraints for the CNF clauses already need to be satisfied simultaneously in an ILP.

Defining the Objective Function. To find an optimal variable assignment, the ILP needs information about how *good* a specific variable assignment is. This is achieved by defining an objective function, which is a formula that determines the optimization score for a given set of variable assignments. The contribution of the individual variables to that score is controlled by their coefficients. Based on the objective function, the integer linear program is able to determines an optimal variable assignment that maximizes/minimizes the objective function, while satisfying all constraints of the feature model.

3 Limitations of Problem-Space Sampling

We argue that technical faults in a configurable system, such as variability induced software bugs, are not located in the problem space, but in the solution space (i.e., in the concrete implementation artifacts). However, validating the solution space based on a conventional problem-space sample can be subject to redundant and missed solution-space interactions, which result in impaired effectiveness and efficiency of the corresponding validation process.

3.1 Redundant Solution-Space Interactions

There are cases where two distinct problem-space interactions lead to the same interaction of implementation artifacts. We denote those interactions as *redundant solution-space interactions*. Considering those redundant solution-space interactions during the sampling process bloats the generated sample, which directly affects its efficiency. For instance, sampling on the software-relevant problem space of our running example requires to cover all valid interactions between software-relevant problem-space features. Two of those interactions are (*ACC, Standard*) and (*ACC, Collapsible*). While those interactions are distinct in the problem-space, they describe the same interaction of software implementation artifacts (*ACC Module, Trailer Hitch Module*). This is because of the presence condition of the implementation artifact *Trailer Hitch Module*, which includes the problem-space features *Standard* and *Collapsible* and is satisfied if either of both features is selected. Thus, this interaction is a redundant solution-space interaction.

3.2 Missed Solution-Space Interactions

A sample that covers all t-wise interactions in the problem-space might still miss t-wise interactions between implementation artifacts. We denote such interactions as *missed solution-space interactions*. Covering all t-wise interactions in the problem space does not cover all t-wise interactions in the solution space. This is because there can be implementation artifacts that require a specific combination of $n > t$ features to satisfy their presence condition. As a t-wise sample only ensures that all valid t-wise feature interactions are covered in the set of configurations, the required combination of $n > t$ features can, but does not have to be included in the sample. For instance, there could be an implementation artifact that is only included in the product, if a specific combination of three features is selected in the configuration. This implementation artifact may require the features ACC and Trailer Assist to be selected while the feature Standard is deselected. Thus, there is a high chance that this implementation artifact and its interactions are not covered in a 2-wise problem-space sample. A problem-space sample that misses particular solution-space artifacts or interactions between them lacks effectiveness, as it cannot be considered as representative. Preventing such missed interactions by increasing t is not reasonable, as there can always be an implementation artifact that requires a specific combination of $n > t$ features. Moreover, a higher t would substantially increase the number of problem-space interactions that need to be covered. This results in a larger sample, while only a few of those interactions are necessary to cover all interactions of implementation artifacts.

4 Generic Solution-Space Sampling

To mitigate the limitations of problem-space sampling regarding the validation of the solution-space, we designed a concept that allows to sample directly on the solution space. In this section, we explain our concept and point out how it eliminates those limitations.

4.1 Feature-Model based Solution-Space Encoding for Multi-Domain Product Lines

As existing sampling algorithms are highly optimized, we designed our concept for solution-space sampling to allow an out-of-the-box reuse of such algorithms to benefit from their performance. To do this, however, we need to encode the solution space with a feature model, as existing sampling algorithms are designed to operate on a feature model, i.e., the problem-space of a product line. This requires to encode the implementation artifacts themselves as well as the dependencies between them, as this is crucial for identifying invalid interactions between implementation artifacts. One approach to achieve this encoding is to construct an *integrated feature model* [16] that encodes the complete product line including the problem-space features, the implementation artifacts, and the configuration knowledge. An integrated feature model can be constructed for any product line for which the problem-space feature model as well as the individual implementation artifacts and their presence conditions are known [16]. To sample on a multi-domain product line, we extend the concept of integrated feature models to construct a *multi-domain integrated feature model* that encodes the solution space of different domains. Therefore, we structure the multi-domain integrated feature model into different sub-trees, one for the problem space and one dedicated sub-tree for each domain-specific solution space. While the problem space sub-tree corresponds to the original problem-space feature model (Figure 2), the solution space sub-trees are created step-wise from the individual domain-specific solution spaces as follows:

To encode the solution space of a domain, each contained implementation artifact is represented by a dedicated *solution-space feature* which is added to the solution space sub-tree of its respective domain. The solution-space features are modeled as optional features without any hierarchy, as there are commonly no explicit dependencies between implementation artifacts. Instead, the dependencies between implementation artifacts result implicitly from their presence conditions and dependencies between problem-space features. For instance, the implementation artifacts *Trailer Hitch Module* and *Trailer Assist Module* can only be present in a product together due to their presence conditions and the constraint $Trailer\ Assist \Rightarrow Standard \vee Collapsible$.

To encode those dependencies, the integrated feature model defines a constraint for each implementation artifact that expresses its presence condition. Such constraint is called *solution-space constraint* [16] and has the structure

$$a \Leftrightarrow p(a)$$

for an implementation artifact a and its presence condition $p(a)$. The integrated feature model that encodes the software domain of the running example is shown in Figure 2.

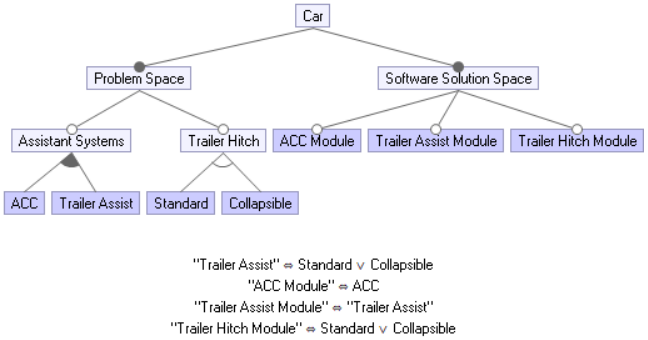

Figure 2. Integrated Feature Model of the Software Domain

4.2 Simplifying the Integrated Feature Model

A problem-space feature can be mapped to multiple implementation artifacts, such as multiple components across different domains that implement its functionality. As the integrated feature model represents each implementation artifact individually with a dedicated solution-space feature, such cases can lead to a large integrated feature model whose number of solution-space features exceeds the number of problem-space features by orders of magnitude. Since the number of features drastically increases the number of interactions that need to be covered, a large integrated feature model affects the sampling time negatively. To mitigate this, we apply two integrated feature-model simplifications that we proposed in earlier work [16]. Those simplifications eliminate solution-space features and constraints without altering the original variability. Both simplifications are based on identifying *atomic sets* [35] of solution-space features.

An atomic set is a set of features that show an equal selection behaviour. This means that selecting any feature in an atomic set also requires the selection of all other features in the atomic set, while deselecting any feature also requires the deselection of all others. Such atomic sets can emerge from the hierarchical dependencies in the feature model (i.e., a mandatory child feature forms an atomic set with its parent feature) or cross-tree constraints. In the context of integrated feature models, an atomic set identifies solution-space features with an equal or equally behaving presence condition. In our running example, this applies to the solution-space features *Trailer Assist Module* and *Trailer Hitch Module* due to their presence conditions and the cross-tree constraint $Trailer\ Assist \Leftrightarrow Standard \vee Collapsible$. Due to the equal selection behaviour, an atomic set can be treated as a single feature during sampling. This information is used to restructure and simplify the integrated feature model with two simplification steps.

Step 1: Eliminating Solution-Space Constraints [16]. For each atomic set of solution-space features, one contained solution-space feature is picked as the *representative feature* for this atomic set. All other features in the atomic set are

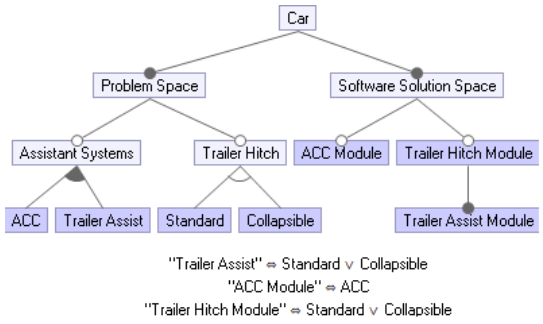

Figure 3. Simplified Integrated Feature Model after Step 1

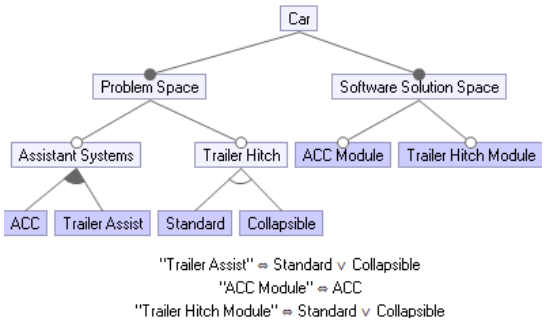

Figure 4. Simplified Integrated Feature Model after Step 2

moved below the representative feature as mandatory child features. As this hierarchical relation preserves the atomic set dependencies, the solution-space constraints of the non-representative solution-space features are no longer required and can be removed from the integrated feature model. For an atomic set of n solution-space features, this simplification eliminates $n - 1$ solution-space constraints from the integrated feature model while maintaining the original variability of contained solution-space features. Even though this simplification omits the original structure of the multi-domain integrated feature model by mixing solution-space features of different domains into a common sub-tree, we reuse it for solution-space sampling.

Step 2: Eliminating Solution-Space Features [16]. Due to the atomic set relation, non-representative features behave exactly like their representative features selection-wise. Therefore, it is sufficient to only consider the interactions between representative features during sampling, as those interactions also cover all interactions between non representative features. As the non-representative features are not considered during sampling, they do not need to be kept in the integrated feature model. Thus, we can also apply the second simplification, which removes all non-representative solution space features from the integrated feature model. This simplifies the integrated feature model and thereby simplifies its corresponding propositional formula that needs to be processed during sampling. However, this simplification also causes overhead as it requires an external mapping between representative and non-representative features that indicates, which non-representative feature is represented by which representative feature. The mapping is required to extend the partial products in the resulting sample with the features represented by the contained features.

4.3 Sampling the Solution-Space

The resulting multi-domain integrated feature model encodes the implementation artifacts and their dependencies. Thus, it allows to reuse existing sampling algorithms to sample on the solution space of included domains. This can be done by sampling only on the subset of representative solution-space features. This functionality is provided by existing sampling algorithms such as YASA [21]. In doing so, only unique interactions between implementation artifacts are considered and it is ensured that all implementation artifacts and their interactions are covered. Table 1 shows the samples resulting from sampling on the software-relevant problem space and sampling on the software solution-space. It can be seen that the software solution-space sample is **33 %** smaller compared to the software-relevant problem-space sample while covering all solution-space interactions.
The modularity of the integrated feature model allows to include arbitrary subsets of implementation artifacts such as those of a particular domain. This is a central benefit of our concept, as it allows to compute samples that are tailored to particular tests scenarios, such as for different integration test levels [30]. Applying solution-space sampling to a particular part of a multi-domain product line, such as the software domain, provides a set of partial products that can be used to validate the particular domain in isolation. However, it may be necessary to also validate the domain in the context of a complete product.

Table 1. Difference in Sample Size between Problem- and Solution-Space Sampling for the Software Domain

2-Wise Software-Relevant Problem-Space Sample
1. { }
2. {ACC}
3. {Trailer Assist, Standard}
4. {Trailer Assist, Collapsible}
5. {ACC, Trailer Assist, Standard}
6. {ACC, Trailer Assist, Collapsible}

2-Wise Software Solution-Space Sample
1. { }
2. {ACC Mod.}
3. {Trailer Assist Mod., Trailer Hitch Mod.}
4. {ACC Mod., Trailer Assist Mod., Trailer Hitch Mod.}

5 Configuration Completion

Testing a particular domain in a complete product is crucial for ensuring the functional correctness of this domain in a realistic environment. There are different approaches to derive a complete product from a partial product. First, the partial product can be extended with implementation artifacts of other domains to assemble a complete car. Alternatively, the partial product can be used to derive a matching problem-space configuration that describes a complete car. We decided for the latter, as this approach produces a configuration that is compatible with other processes in the product line's tool chain. For instance, the configuration could be used as input for the standard production process of a car.

5.1 The Optimization Problem

One possibility to find a matching complete configuration for a given partial product is boolean constraint propagation (BCP). BCP derives the assignments of unassigned features based on their dependencies to already assigned features. For instance, if the partial product contains the ACC Module, BCP would select the feature ACC as well, as this is required due to the constraints in the integrated feature model. If no further feature assignment can be derived from already assigned features, BCP proceeds to assign an arbitrary value (true / false) to an unassigned feature and starts from the beginning. However, there can be multiple matching problem-space configurations for a given partial solution-space product. For instance, the partial product

{*ACC Module*, *Trailer Assist Module*, *Trailer Hitch Module*}

contained in the software solution-space sample has two matching complete configurations:

{*ACC*, *Trailer Assist*, *Standard*}

{*ACC*, *Trailer Assist*, *Collapsible*}

As the matching configurations may differ in non-functional properties, such as their total price, selecting one of the matching configurations becomes an optimization problem.

5.2 Finding Optimized Configurations

There are various possible optimization criteria that can be used to compute an optimized complete configuration for a given partial product. For instance, one could aim to find the cheapest matching car configuration for a given software configuration to reduce the testing costs. However, finding such an optimized configuration is difficult, as a greedy selection of the cheapest individual problem-space features may not be possible due to problem-space constraints. Thus, to find an optimized complete configuration for a given partial product, we need to determine the optimal selection of problem-space features so that the target optimization criterion is maximized or minimized, while all constraints are considered. We do this by translating the integrated feature model into an integer linear program as described in Section 2.3. To encode how an individual feature selection contributes to the optimization score, we assign a coefficient, e.g. the price of the problem-space feature, to its respective ILP variable in the objective function. Moreover, we need to ensure that the solution-space features contained in the partial product, for which we aim to find the optimized complete configuration, remain selected during the optimization process. This is important, since the selection of those solution-space features limits the valid selections of problem-space features due to constraints between them. For instance, a partial product that contains the solution-space feature *ACC Module* implicitly forces the selection of the problem-space feature *ACC* due to their presence condition. We ensure that those features remain selected by changing the value range of their respective ILP variables from [0,1] to [1] so that those variables can only take on the value 1 (selected). Solving the resulting ILP returns assignments for all features contained in the integrated feature model. We filter those assignment for problem-space features that are assigned with the value 1 (selected) to obtain the optimized complete configuration for the target partial product.

6 Evaluation

To evaluate our concept for solution-space sampling, we apply it to six real-world, automotive product lines that serve as subject systems in our evaluation. In this section, we explain our evaluation setup and define the research questions that we aim to answer.

6.1 Experimental Setup and Execution

We use six automotive product lines provided by our industry partner as subject systems for our evaluation. Each of those subject systems is an actual multi-domain product line with a single problem-space feature model that describes the configurability of all domains. As we were only provided with the data of the software domain, i.e., software artifacts and their presence conditions, our evaluation focuses on the software domain. Table 2 shows the characteristics of the subject systems. The problem-space characteristics are given by the columns #Fs (total number of problem-space features), #SFs (number of software-relevant features, Definition 2.1) and #CTCs (total number of cross-tree constraints). The software solution-space characteristics are given by the columns #As (total number of software artifacts) and #RAs (total number of remaining software artifacts after applying the simplifications explained in Section 4.2). The immense impact of the simplifications regarding the elimination of solution-space features stems from the fact that there are many software artifacts that belong to the same problem-space feature, i.e., have the same presence conditions and therefore form an atomic set. Moreover, a large portion of software artifacts are present in each product, thus also forming an atomic

Table 2. Characteristics of the Subject Systems

	Problem Space			Solution Space	
	#Fs	*#SFs*	*#CTCs*	*#As*	*#RAs*
Automotive 06	462	172	606	6,853	69
Automotive 07	436	152	569	3,369	34
Automotive 08	591	236	906	11,764	176
Automotive 09	611	247	2,615	4,826	201
Automotive 10	526	184	1,085	10,532	163
Automotive 11	681	285	4,659	15,132	353

set. We implemented our concept for solution-space sampling using the FeatureIDE library v3.7.2[1] [27]. This library provides many functionalities required for our concept such as constructing feature models and identifying atomic sets. Moreover, this library provides an implementation of the YASA sampling algorithm [21] that we use in our evaluation. To implement the optimizations, we used the Google OR-Tools library v.9.2.9972[2] that provides a Java API[3] for defining an ILP. We performed the evaluation on a conventional notebook with an Intel i7-9850H @2,60 GHz, 32 GB RAM and Windows 10 64 Bit. Due to closed source restrictions, we cannot provide the implementation and the input data.

6.2 Research Questions

We define three research questions as guideline for our evaluation. In this section, we explain those research questions and point out how we aim to answer them.

RQ1: *How do the sample properties differ between problem-space and solution-space sampling for the software domain?*
To evaluate the difference between problem and solution-space sampling, we apply both sampling approaches to the software domain of the six subject systems. In doing so, we capture the number of valid interactions in the problem and solution space as this is a central indicator for the complexity that the sampling process needs to deal with. Moreover, we capture the size of both samples and determine the number of solution-space interactions that are missed by the problem-space sample.

RQ2: *How does the sampling time differ between problem-space and solution-space sampling for the software domain?*
Our solution-space sampling concept requires computational overhead for constructing the integrated feature model. Additional overhead is required for the atomic set computation when applying the simplifications introduced in Section 4.2. Thus, we evaluate how the total run time required for the sampling process differs between conventional problem-space sampling and solution-space sampling. Moreover, we expect the simplifications to affect the sampling time, as the simplifications drastically reduce the number of solution-space features and constraints that need to be considered during solution-space sampling. Thus, we aim to evaluate how those simplifications affect the sampling time by performing solution-space sampling with and without enabled simplifications.

RQ3: *How does the average installation rate differ between automatically completed and optimized configurations?*
Our solution-space concept includes to compute optimized complete configurations for the partial products contained in the software solution-space sample. In our evaluation, we determine how the optimization score of the completed configurations differ between automatically and optimized completed configurations. To do this, we use data for the installation rate of each problem-space feature as optimization criteria. We then apply the ILP optimization to find a matching problem-space configuration with the highest average installation rate for each partial product contained in the software solution-space samples. Here, it is of special interest how well the optimizations works in the context of the already selected solution-space features contained in the partial products.

6.3 Results

In the following sub-section, we show the results that we obtained by executing the previously described experiments and highlight our most prominent findings.

Total Number of Interactions. We captured the number of valid interactions that need to be covered in a 2-wise sample for the software-relevant problem space and for the software solution-space. The numbers for the software solution-space were calculated based on the simplified integrated feature models. The results are visualized in Figure 5. In general, the data shows that there are more interactions in the software-relevant problem space for two subject systems while there are more 2-wise interactions in the software solution-space for the other four subject systems. However, the actual difference in the number of 2-wise interactions varies strongly between the subject systems. In detail, we found the maximum difference for the number of interactions in Automotive 07 with a total of **13,526** interactions in the software-relevant problem space, but only **2,123 (16 %)** interactions in the software solution-space. In contrast, the minimum difference was found for Automotive 09, with a total of **77,641** interactions in the software solution-space and **71,694 (92 %)** in the software-relevant problem space. Overall, the average relative difference in the total number of valid interactions amounts to **56.5 %**.

[1] https:/featureide.github.io
[2] https:/developers.google.com/optimization
[3] https:/github.com/google/or-tools/releases

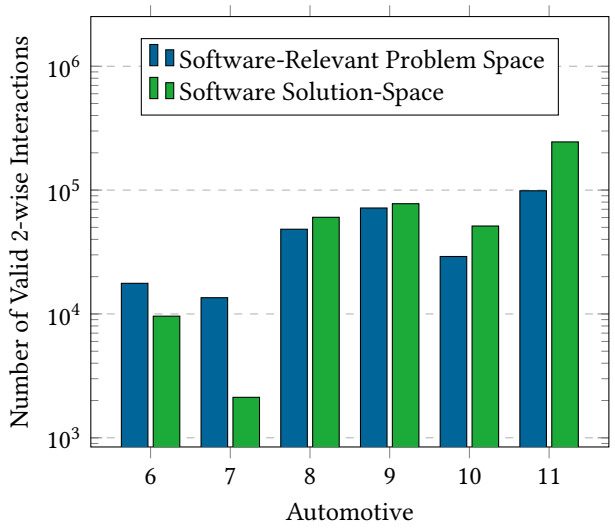

Figure 5. 2-wise Interactions in the Software-Relevant Problem Space and Software Solution-Space per Subject System

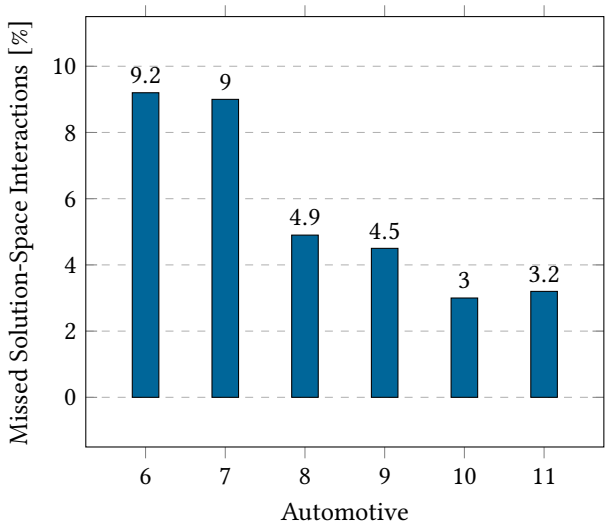

Figure 6. 2-wise Solution-Space Interactions that are missed by the 2-wise Problem-Space Sample per Subject System

Missed Solution-Space Interactions. We determined the number of 2-wise interactions in the software solution-space that are not covered by the 2-wise sample for the software-relevant problem-space. We then calculated the ratio between the missed solution-space interactions and the total number of solution-space interactions. Figure 6 shows the results of this analysis. In general, we found that there are missed software solution-space interactions in the software-relevant problem-space sample for each subject system. The maximum ratio of those missed interactions was found for Automotive 06 with a ratio of **9.2 %**, while the minimum ratio was found for Automotive 10 with a ratio of **3 %**. The average ratio of missed interactions across all subject systems is **5.6 %**. In absolute values, the maximum of missed solution-space interactions was found for Automotive 11 with a total of **7835** missed interactions, while the minimum absolute number was found for Automotive 07 with a total of **191** missed interactions.

Sample Size. To further evaluate solution-space sampling, we captured the size of the resulting samples for the software-relevant problem space and the software solution-space for each subject system. The results are shown in Figure 7.
It can be seen that the software solution-space sample is substantially smaller than the software-relevant problem-space sample for five out of six subject systems. Only for Automotive 10, the sample for the software solution-space is about **10 %** larger compared to the sample for the software relevant problem-space. The maximum relative difference in the sample size was found for Automotive 07, where the sample for the software-relevant problem space contains a total of **32** configurations, while the sample for the software solution-space only contains **14** partial products. This amounts to a relative sample-size reduction of **56 %**. The minimum relative sample-size reduction, in contrast, was found for Automotive 11, where the sample for the software-relevant problem space contains **218** configurations, while the sample for the software solution-space contains **149** partial products, which amounts to a relative sample-size reduction of **30 %**. Overall, the average relative sample-size reduction of solution-space sampling amounts to **32 %** across all subject systems.

Sampling Time. To reason about the sampling time, we captured the run time of the problem- and solution-space sampling processes. To gain detailed information about the

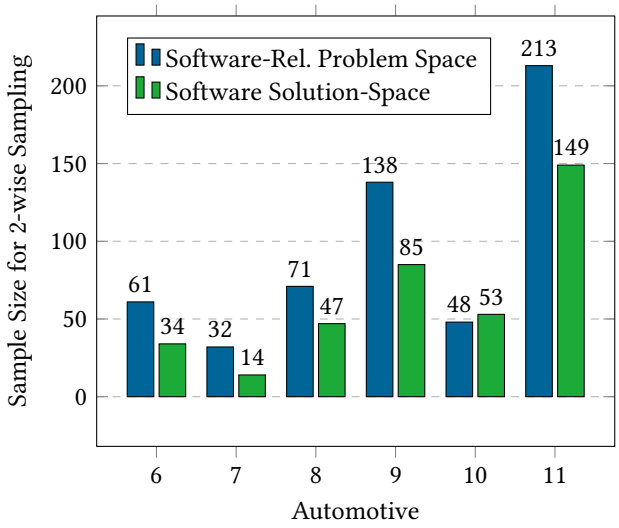

Figure 7. Sample Size for the Software-Relevant Problem Space and the Software Solution-Space per Subject System

Table 3. Run Time of the Single Steps of Problem- and Solution-Space Sampling in Seconds (> 3,600.00 s Indicates a Timeout).

	Problem-Space Sampling			Solution-Space Sampling			
	Identifying Relevant Features	*Sampling*	*Total*	*Constructing Integrated FM*	*Simplifications*	*Sampling*	*Total*
Automotive 06	0.13 s	1.23 s	1.36 s	13.53 s	yes - 12.26 s no - 0 s	0.98 s 357.98 s	26.78 s 371.51 s
Automotive 07	0.09 s	0.78 s	0.87 s	4.43 s	yes - 2.70 s no - 0 s	0.51 s 12.15 s	7.65 s 16.58 s
Automotive 08	0.71 s	2.11 s	2.82 s	118.49 s	yes - 38.94 s no - 0 s	3.45 s > 3,600.00 s	160.89 s > 3,718.50 s
Automotive 09	0.04 s	4.35 s	4.38 s	5.66 s	yes - 9.19 s no - 0 s	6.77 s > 3,600.00 s	21.63 s > 3,605.70 s
Automotive 10	0.40 s	1.35 s	1.76 s	69.70 s	yes - 32.24 s no - 0 s	2.22 s > 3,600.00 s	104.17 s > 3,669.70 s
Automotive 11	0.41 s	13.65 s	14.07 s	90.27 s	yes - 66.99 s no - 0 s	27.80 s > 3,600.00 s	185.07 s > 3,690.27 s

run time behavior of both sampling approaches, we did not only capture the total run time, but also the run time of the individual steps that are required for problem- and solution-space sampling on the software domain. The results are shown in Table 3. The data shows that, in terms of total run time, solution-space sampling takes substantially longer than problem-space sampling for each subject system. In detail, we found that solution-space sampling takes between **5** (Automotive 09) and **60** (Automotive 10) times longer compared to problem-space sampling. The data also shows that this increase results from the computational overhead that is required for constructing the integrated feature model and its simplification. Those steps are responsible for **69 %** (Automotive 09) to **98 %** (Automotive 08 and Automotive 10) of the total run time of the complete solution-space sampling process. However, the run time data also shows that omitting the simplification causes the sampling step to take substantially longer or to even run into our defined timeout.

Optimized Product Completion. We chose three of the subject systems for applying the optimized configuration completion. The results are shown in Figure 8. The configuration with the highest average installation rate for each subject system, computed by the ILP, determines the upper bound for the optimization scores and is visualized as support line in the charts in Figure 8. In general, the data shows that that the optimization scores vary strongly for the individual partial products for which the optimized configuration was computed. In detail, the average optimization score of the automatic completion ranges from **43.78** (Automotive 09) to **45.28** (Automotive 06). In contrast, the average optimization score of the maximized completion ranges from **46.88** (Automotive 08) to **48.86** (Automotive 06). This amounts to an average optimization score increase of **3.33** compared to the automatically completed configurations.

6.4 Discussion

In this section, we summarize the results obtained in our evaluation to answer our previously defined research questions.

RQ1: *How do the sample properties differ between problem-space and solution-space sampling for the software domain?* Our evaluation has shown that there is a substantial difference between solution- and problem-space sampling. First, the results show that there are drastically more 2-wise interactions in the software solution-space compared to the software-relevant problem space for the majority of the subject systems. This is because there is no 1:1 mapping between problem-space features and implementation artifacts. Moreover, we found that conventional problem-space sampling did not cover all 2-wise solution-space interactions. The detailed analysis in our evaluation has shown that 2-wise problem-space sampling covered at most 97 % of all 2-wise solution-space interactions. Investigating some of the missed interactions has shown that they result from implementation artifacts that require a specific selection of $n > 2$ features. Moreover, the evaluation has shown that solution-space sampling produces up to 56 % smaller samples compared to conventional problem-space sampling. Only in a single case, the solution-space sample was slightly larger, which we assume to be due to the missed solution-space interactions that had to be covered additionally.

RQ2: *How does the sampling time differ between problem-space and solution-space sampling for the software domain?* Our evaluation has shown that the sampling time of solution-space sampling is substantially longer compared to the sampling time of conventional problem-space sampling. The detailed analysis of the run time has shown that this increase in run time almost completely results from the computational

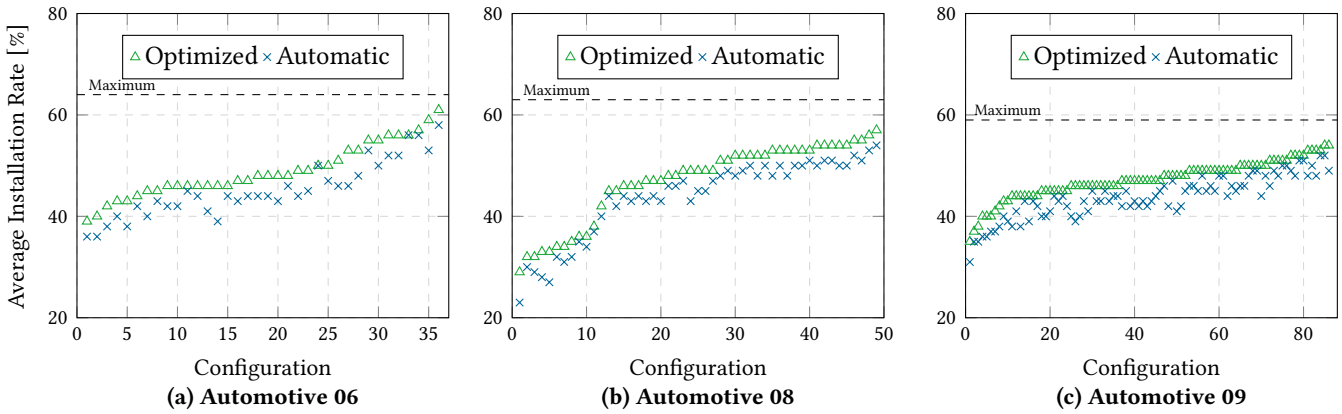

Figure 8. Average Installation Rate of Automatically and Optimized Completed Configurations Sorted by Optimization Score

overhead required for constructing and simplifying the integrated feature model. However, the results also show that the simplifications are crucial for the scalability of solution-space sampling, as omitting the simplifications results in a drastic increase in sampling time which exceeds the time required for the simplifications by a multiple for all subject systems.

RQ3: *How does the average installation rate differ between automatically completed and optimized configurations?*
For almost every partial product, the optimized complete configuration reaches a higher optimization score than its respective automatically completed configuration. However, the actual difference between both scores is not as high as expected, which we assume to be caused by the pre-selected solution-space features. This corresponds to our observation that the reachable optimization score strongly depends on the particular partial product and its contained solution-space features, as those strongly limit the optimization possibilities.

6.5 Threats to Validity

The results obtained in our evaluation are subject to different internal and external threats to validity.

Internal Validity. The first threat that may affect the internal validity of our results is the correctness of our implementation. To mitigate this threat, we tested our implementation by applying it to small, artificial product lines for which the results could be manually verified. We compared the manually retrieved results with the results provided by the implementation and ensured that they match. Moreover, we used the FeatureIDE library for building the integrated feature model, for computing the atomic sets and for performing the actual sampling, as this library is frequently used in research and industry [1, 15]. Moreover, to handle the complexity of the input data, we discussed it in detail with the respective experts from our industry partner.

External Validity. We were only able to apply our concept of solution-space sampling to proprietary product lines. This is because we require the data of large multi-domain product lines, which are, to our knowledge, not publicly available. Moreover, our optimization concept requires specific data, such as the installation rates of individual problem-space features, which is also not available for publicly accessible product lines. Thus, even though we were able to show the benefit and scalability of solution-space sampling for the analyzed proprietary subject systems, we cannot state that those results are representative for other product lines.

7 Related work

Our concept for solution-space sampling is related to other work in the context of sampling and product lines. In this section, we list the related work and point out how it differs from our contribution.

Multi-Domain Product Lines. Multi-domain product lines and their challenges have already been addressed in existing work. Kowal et al. [20] developed a concept that helps to maintain multi-domain product lines by explaining feature-model anomalies that arise due to constraints between domains. Ananieva et al. [2] have presented an approach to reveal implicit dependencies between features of different domains. However, both contributions do not consider the solution space and do not focus on sampling.

Feature-Model Sampling. Krieter et al. [21], Al-Hajjaji [1], Garbin et al. [14] and Johansen et al. [17] have developed algorithms for computing t-wise interaction samples for feature models. In our work, we do not propose a new sampling algorithm but reuse the algorithm developed by Krieter et al. Devroey et al. [12] presented a concept that prioritizes the configurations based on the behaviour of the products. Therefore, the authors use statistical models to prioritize configurations based on the likelihood of their executions.

Solution-Space Validation. Considering the actual implementation artifacts during product-line validation has also been proposed in existing work. In previous work, we have introduced integrated feature models that allow to compute the number of derivable products [16]. We extended this concept for multi-domain product lines and reused it for solution-space sampling. Papadakis et al. [29] have proposed a concept to validate configurable software by applying mutation testing to the variable code blocks and combining it with combinatorial interaction testing. In contrast to our work, the authors only considered the interactions of problem-space features. Moreover, Tartler et al. [32] have introduced configuration coverage, an approach to compute a set of configurations that allows to type check each possible configuration of variable code blocks. Moreover, Tartler et al. [31] have proposed their tool *vampyr* which computes a set of configurations that allow to type check all variable code blocks in system software with compile-time variability (e.g., the Linux Kernel). In contrast to our work, however, they do not consider the interactions of implementation artifacts. Kim et al. [19] present a concept to reduce the sample size by eliminating features that do not alter the behaviour of the software artifacts under test and therefore can be omitted during sampling. A similar approach has been presented by Shi et al. [30], who apply symbolic execution to the variable code blocks to find actual feature interactions that need to be considered during sampling and testing. In contrast to our work, those contributions aim to validate the correctness of the variable source code of a product line statically.

Optimized Configuration Selection. Integer linear programming has also been used in other work in the context of product lines and feature models. Baller et al. [5] have presented a concept to select a minimum test set for a set of configurations under test derived from a product line using ILP. Furthermore, Bernavides et al. [8] proposed extended feature models that support features attributes. The authors used such extended feature models in combination with constraint programming for different feature-model queries such as finding the optimal configuration for a given attribute. Instead of finding a single optimal configuration in the problem-space of a product line, we used the ILP optimization to find an optimized configuration for a set of pre-selected features.

8 Conclusion and Future Work

In this paper, we have introduced solution-space sampling, which allows to sample on the solution space of a multi-domain product line by encoding the solution space with a feature model. Our evaluation has shown that solution-space sampling produces smaller samples for the vast majority of the subject systems, while ensuring that all solution-space interactions are covered. However, there is a trade-off in terms of sampling time.

Constructing the set of sampled products for a multi-domain product line requires manual effort, due to physical components. Thus, in future work, we aim to design a sampling concept that considers and reduces this manual effort by reducing the number of different hardware topologies in the sampled products.

Disclaimer

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

References

[1] 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.

[2] 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.

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

[4] M Ben Ayed, Lilia Zouari, and Mohamed Abid. 2017. Software in the loop simulation for robot manipulators. *Engineering, Technology & Applied Science Research* 7, 5 (2017).

[5] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. 2014. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 303–312.

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

[7] Andreas Bayha, Franziska Grüneis, and Bernhard Schätz. 2012. Model-based software in-the-loop-test of autonomous systems. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*. Citeseer, 1–6.

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

[9] 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.

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

[11] 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.

[12] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. 2014. Towards Statistical Prioritization for Software Product Lines Testing. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 10, 10:1–10:7 pages.

[13] Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)* 53 (2011), 2–13. Issue 1.

[14] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)* 16, 1 (2011), 61–102.

[15] 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.

[16] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the Variability Mismatch Between Problem and Solution Space. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*. ACM, 11 pages.

[17] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 46–55.

[18] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 25–32.

[19] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2010. Eliminating Products to Test in a Software Product Line. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 139–142.

[20] Matthias Kowal, Sofia Ananieva, Thomas Thüm, and Ina Schaefer. 2017. Supporting the Development of Interdisciplinary Product Lines in the Manufacturing Domain. *World Congress of the International Federation of Automatic Control (IFAC)* 50, 1 (2017), 4336–4341.

[21] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 4, 10 pages.

[22] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 31–40.

[23] 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.

[24] 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.

[25] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 176–187.

[26] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654.

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

[28] 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.

[29] Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. 1–10.

[30] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 270–284.

[31] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proc. USENIX Annual Technical Conference (ATC)*. USENIX Association, 421–432.

[32] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review* 45, 3 (2012), 10–14.

[33] 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.

[34] 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.

[35] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. *Formal Methods and Software Engineering* (2004), 115–130.

Received 2022-08-12; accepted 2022-10-10