Consistency Is Key: Can Your Product Line Realise What It Models?

Philip Ochs
philip.ochs@kit.edu
Karlsruhe Institute for Technology
Germany

Tobias Pett
tobias.pett@kit.edu
Karlsruhe Institute for Technology
Germany

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

Abstract

Nowadays, automotive systems are modelled as cyber-physical product lines. However, often it is not clear whether a modelled configuration can be realised as a physical product. The combination of software- and hardware-artefacts harbours the risk of non-functioning products due to incompatible resource demands and provisionings of the components installed (realisability), thus can result in high financial loss for manufacturers. With new business models, such as over-the-air updates, they also face this risk for products already in the field, because it remains unclear whether a vehicle is still functioning after an update where resource demands have changed (update-ability). Manually analysing realisability and update-ability is infeasible in practice, as the number of product variants in a product line grows combinatorially (with respect to the number of configuration options). In this paper, we approach this challenge by proposing a novel baseline approach for the analysis of realisability and update-ability in automotive cyber-physical systems. We formally model resource demands and resource provisionings as well as the construction of a resource allocation problem per product variant based on constraint satisfaction problems. In the evaluation, we apply our method to an automotive case study by modelling realisation artefacts and investigate its feasibility and performance. Our results show that a non-realisable configuration of the product line can be identified in 206 ms median time. Finally, we discuss limitations and extensions of our ongoing work.

CCS Concepts

• **Software and its engineering** → **Software product lines**; *Model-driven software engineering*; Software configuration management and version control systems.

Keywords

product line engineering, cyber-physical systems, product variant analysis, product line consistency, realisability analysis

ACM Reference Format:
Philip Ochs, Tobias Pett, and Ina Schaefer. 2024. Consistency Is Key: Can Your Product Line Realise What It Models?. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3687812

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687812

1 Introduction

Modern cyber-physical systems combine hardware and software components and are typically highly configurable. This introduces complexity with respect to the interplay between the used hardware and the deployed software in context of possible variants. The complexity can be managed by designing the system as a product line, where a collection of customised products can be built by selecting configuration options (i.e., features) [2, 5]. However, the variability contained in a product line leads to a potentially large number of possible products, e.g., 33 optional features lead to more than 8 billions of possible variants [16]. The system's manufacturer has to know which of these variants can be offered, built and sold. In detail, this leads to two use cases as follows:

First, the manufacturer has to identify which combinations of features are valid (problem space), and then which variants are functioning (i.e., which are realisable) in context of the used hardware and the deployed software components (solution space). For example, in the problem space, a car can have either a navigation system or a multimedia system or both at the same time. However, in the solution space, the provided resources can be allocated for either of these two features, but do not satisfy the demands of both features together. Thus, a configuration can be valid for the combination of features, but not realisable in practice as inconsistencies between modelled and actual variability yield incompatible artefacts in production. We consider a product line with a mismatch between the modelled variability in the problem space and the realisable variability in the solution space as *inconsistent*.

The second use case concerns new business models such as function on demand and over-the-air updates, where software updates are applied to products in the field. This may cause that the products in the field do not work as intended or stop working completely, which can lead to high financial losses for manufacturers. For example, a car's navigation system may be updated to a new version and now has more extensive resource demands. However, some of the cars in the field are equipped with hardware which would not satisfy the resource demands of the updated navigation system version. Therefore, the manufacturer has to identify which product variants may receive an update (i.e., are update-able) so that they are still functioning after deployment.

Both described use cases can be addressed by answering the overall research question: How to ensure the realisability and update-ability of variants in a product line of a cyber-physical system before and after production? Building and analysing all product configurations manually is infeasible in practice, because of the well-known combinatorial explosion problem of product lines [10].

In the literature, a meta-model for describing a generic product line is given by Wittler et al. [28] as an extension to the Unified

Conceptual Model (UCM) of Ananieva et al. [1]. In the solution space, it consists of heterogeneous artefacts representing software and hardware components which demand or provide resources. However, it lacks support for resource analysis. State of the art analysis methods for the problem space of a product line typically address the management of variability [10]. However, they do not cover solution space concerns, such as realisability or update-ability. In the solution space in turn, existing analysis methods are either not fine-grained enough to express solution space constraints on a resource level [13, 14], or are not generic enough to cope with heterogeneous artefacts of cyber-physical product lines [17].

In this paper, we answer the overall research question as follows: We model variability-aware resource demands in the problem space based on the UCM of Wittler et al. [28]. With respect to a single product configuration, we give a formal mapping to derive its concrete resource demands in the solution space. Furthermore, we construct a constraint satisfaction problem for resource allocation in the solution space, which can be processed by constraint satisfaction problem (CSP) solvers. We combine our approach for analysing resource allocation in the solution space with existing problem space analysis methods to address the consistency between variability in problem and solution space. In a baseline evaluation, we extend the solution space of the real-world automotive Body Comfort System (BCS) case study [20] and add resource types, resource provisionings and variability-aware resource demands. We investigate the feasibility of our modelling approach as well as the performance of deciding the consistency of the product line.

In summary, our contributions are:

- A formal model for specification of variability-aware resource demands and their allocation behaviour in cyber-physical product lines.
- A novel product-based analysis method to decide the realisability and update-ability of a configuration as well as the consistency between problem and solution space of a cyber-physical product line based on resource allocation problems.
- An evaluation investigating the feasibility and performance of our work based on the automotive BCS case study [20], which serves as a starting point for further validation.

Our contributions allow to automatically analyse if a specific configuration yields a functioning product which satisfies all resource demands. Furthermore, they enable assessing a product in field to decide if it may receive an over-the-air update, so that it remains functioning after deployment. By combining problem and solution space analysis, we can decide if a cyber-physical system is consistent in its design with respect to modelled and actual variability.

2 Background

This section covers basic terms and concepts about modelling and testing realisability of product lines. First, we introduce product lines and variability, as these are core concepts of this work. Second, we introduce a meta-model for describing product lines formally. Third, we give basic information about constraint satisfaction problems, because we use them for our analysis methods. Finally, we present an automotive running example on which we demonstrate our proposed work throughout this paper.

2.1 Product Lines & Variability

The concept of a *product line* enables the development and production of configurable systems which share a set of commonalities and can be customised with variable functionality [5]. This enables reuse of artefacts and reduction of development costs.

A product line can be divided into problem space and solution space [2]. The *problem space* covers modelling and analysis of theoretical variability aspects (e.g., configuration options), while the *solution space* consists of realisation artefacts (e.g., code implementations) and dependencies and covers aspects of product assembly.

The *feature model* is a central artefact of the problem space and specifies all possible configuration options (*features*) and their dependencies [2]. Features can either be selected or not and relate to other features, for example in terms of optional selection. *Abstract* features are features which do not have realisation artefacts (e.g., only exist for structural purposes). A feature can have *attributes* of various types (numerical, textual etc.) to enrich it with meta-information [3]. We denote a feature and the set of all features in a product line with $f_l \in F$ with $l \in [0, \ldots, |F| - 1]$.

A *configuration* is a concrete selection of features from the feature model. We denote a configuration and the set of all possible configurations in a product line with $C \in \mathcal{CS} := \mathcal{P}(F)$. As an edge case, we do not consider the empty configuration $C_\emptyset$ (i.e., the configuration which does not contain any feature; $\{C_\emptyset\} \cap \mathcal{CS} = \emptyset$). We refer to a configuration which respects all dependencies of the problem space (i.e., of the feature model) as a *valid* configuration and denote the set of all valid configurations of a product line with $\mathcal{VS} \subseteq \mathcal{CS}$. Analogous, we refer to a configuration which respects all dependencies of the solution space (i.e., between realisation artefacts) as a *realisable* configuration and denote the set of all realisable configurations of a product line with $\mathcal{RS} \subseteq \mathcal{CS}$.

2.2 Unified Conceptual Model

The UCM of Ananieva et al. [1] represents a meta-model of a generic product line. In the problem space, it combines concepts of variability in space (e.g., a feature model) and variability in time (i.e., evolution of the system). The solution space represents implementation artefacts as *fragments*, which are derived from concrete product variants. Wittler et al. [28] extended the UCM by a meta-model of the solution space, where fragments are replaced by components which are mapped from features. In detail, the solution space is described by software components, which demand resources, and hardware components, which provide resources. A resource has an associated resource type with two properties: The property `boundaryType` defines if a provided resource quantification has to be (1) above a lower, (2) below an upper or (3) equal to a quantified resource demand. Furthermore, a resource can be exclusive (property `isExclusive`), which defines whether a resource is allocated for at most a single component. This behaviour can be enforced for safety-critical software components that should be deployed to one hardware component exclusively. We denote software components and the set of all software components with $sw_i \in SW$ with $i \in [0, \ldots, |SW|-1]$ as well as hardware components and the set of all hardware components with $hw_j \in HW$ with $j \in [0, \ldots, |HW|-1]$. We denote a resource type and the set of all resource types with $rt_k \in RT$ with $k \in [0, \ldots, |RT|-1]$.

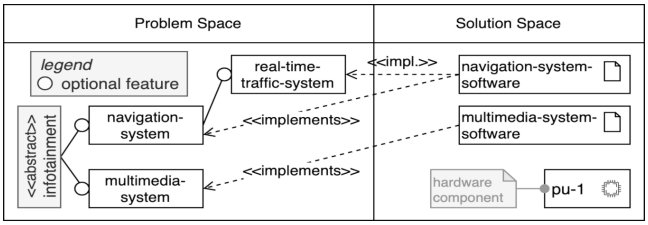

Figure 1: Problem and solution space of the infotainment system running example used in this paper.

2.3 Constraint Satisfaction Problems

In propositional logic, a CSP is an approach to express a logical problem where a number of variables adopt allowed values in order to satisfy constraints [23]. In detail, a CSP consists of a set of variables and a set of constraints. Each constraint consists of the specification of variables which occur in the constraint and a relation, e.g., a propositional logical formula. For simplicity, we typically omit the explicit specification of participating variables and directly give a propositional logical formula for the constraint. We denote constraints and the set of all constraints with $T \in \mathcal{T}$. A set of variables has a set of possible values, i.e., a *domain*, for example the boolean domain. A CSP is *satisfiable* if there exists an assignment of domain values to the variables so that all constraints are satisfied. CSPs are typically solved by commercial off-the-shelf (COTS) tools, depending on the complexity of the stated problem (e.g., boolean propositional logic [12] or satisfiability-modulo-theories [7]).

2.4 Running Example

In this paper, we use an automotive product line of a vehicle's infotainment system as a running example to illustrate our concepts and procedures. It is shown in Figure 1 and represents an instantiation of the UCM. In the problem space, the feature model consists of the optional features `multimedia-system` and and `navigation-system`. The feature `navigation-system` can optionally be extended by the feature `real-time-traffic-system`. In the solution space, the software component `navigation-system-software` implements the functionality of the features `navigation-system` and `real-time-traffic-system` while the software component `multimedia-system-software` implements the functionality of the feature `multimedia-system`. The software artefacts can be deployed on one hardware component (processing unit `pu-1`).

Table 1a shows the defined, simplified resource types and their properties in the running example: RAM describes a non-exclusive, lower boundary and the system response time a non-exclusive, upper boundary resource type.

name	isExclusive	boundaryType
RAM	**False**	LOWER
Response Time	**False**	UPPER

(a)

isAdditive
True
False

(b)

Table 1: Specification of resource types in the running example. (a) Exclusivity and boundary type [28] and (b) additivity.

3 Modelling Variability-Aware Resource Demands

The solution space of a product line of a cyber-physical system consists of hardware and software artefacts which are related to product configurations in the problem space. To model these solution space artefacts, we use the UCM of Wittler et al. [28]. In the problem space, it consists of a feature model. In the solution space, dependencies are modelled between hardware components providing resources and software components demanding resources. Thus, these dependencies describe a resource allocation problem related to a specific configuration from the problem space. To solve this allocation problem, we first introduce how the allocation behaviour of resources can be specified by resource type properties. Then, we describe how we can model resource demands independently of configurations and based on feature-attributes (i.e., feature-defined resource demands). Finally, we define mappings between configuration-independent and configuration-specific resource demands.

3.1 Specifying Resource Allocation Behaviour

Wittler et al. [28] modelled resources to have an associated resource type. We use resource types to define the allocation behaviour of resources via three resource type properties as follows.

Additivity. We introduce the boolean resource type property `isAdditive` to define whether or not multiple defined resources of the same type sum up in value. We extend the defined resource types of the running example with this property as shown in Table 1b: The resource type for RAM is additive, because if one software component demands 3 GB of RAM and another software component demands 1 GB of RAM, both deployed together demand 4 GB of RAM in total. The resource type for the system response time is non-additive, because demands of this resource type semantically do not sum up over multiple software components. In this case, we assume that only the strictest demand must be satisfied in order to satisfy all others.

Exclusivity & Boundary Type. As second and third resource type property, we use the attributes `isExclusive` and `boundaryType` as presented in Section 2. Exclusivity specifies whether or not a resource is allocated for (a maximum of) a single component, for example safety-critical components. The boundary type defines, if a provided resource quantification must be above a lower, below an upper or exact to a resource demand. Figure 2 shows the resulting solution space model based on the UCM of Wittler et al. [28].

3.2 Describing Feature-Defined Resource Demands

The resource demands of a configuration highly depend on the concretely selected features. In the running example, both features `navigation-system` and `real-time-traffic-system` are implemented by the software component `navigation-system-software`. Therefore, the concrete resource demands of this software component depend on the customer's choice of selecting (1) none of these two features (then, software component `navigation-system-software` is not in use), (2) only the feature `navigation-system` or (3) both.

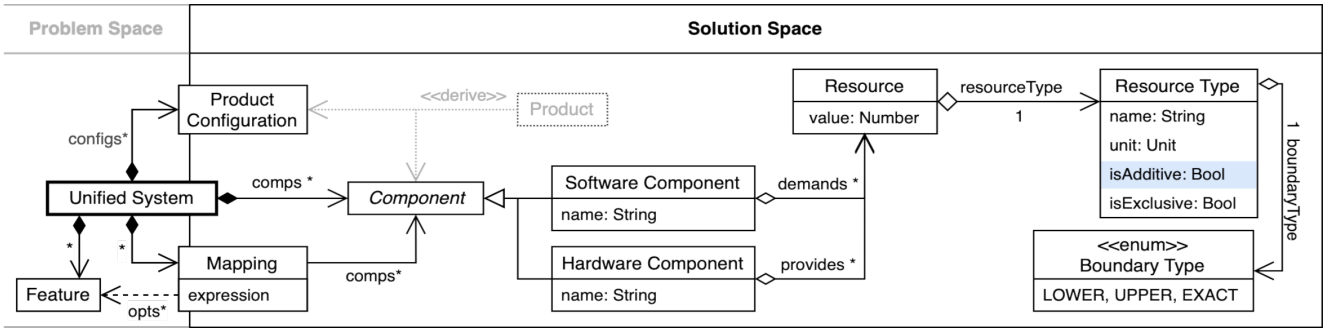

Figure 2: Extended (blue) solution space model based on the UCM of Wittler et al. [28].

To allow the decoupling between configuration-independent and configuration-specific resource demands, we model a feature-defined resource demand as a feature attribute [3] in the problem space. It expresses a resource demand value concerning a software component sw_i in context of a resource type rt_k specified by a feature f_l and is denoted by $fd_{i,k,l}$. In Figure 3, we visualise a feature-defined resource demand as an edge from the feature to the demanded resource type and consecutively to the software component it concerns. The edge weight represents the value of the feature-defined resource demand. The feature `multimedia-system` demands 1 GB of RAM and a maximum system response time of 100 ms to the software component `multimedia-system-software`. The feature `navigation-system` demands 3 GB of RAM and a maximum system response time of 50 ms to the software component `navigation-system-software`. The feature `real-time-traffic-system` demands 1 GB of RAM and a maximum system response time of 30 ms to the software component `navigation-system-software`.

3.3 Mapping Feature-Defined Resource Demands

For a concrete configuration, solution space artefacts (e.g., software components) are derived from a feature selection in the problem space. After we introduced feature-defined resource demands in the problem space, we now map these demand specifications to configuration-specific resource demands of software components in the solution space.

The mapping operates on a set of feature-defined resource demands for a specific software component sw_i in context of a specific resource type rt_k ($\{fd_{i,k,0}, \ldots, fd_{i,k,|F|-1}\}$). It maps this set to a single value which describes the configuration-specific resource demand of the software component sw_i in context of the resource type rt_k. Thus, it is applied per defined software component and per defined resource type. Formally, each mapping distinguishes between five cases along the resource type properties of the given resource type as follows: (1) If the resource type is additive, we sum up the given feature-defined resource demands. Else, depending on the boundary type, we (2) take the maximum, (3) minimum or (4) a random value in the set (assuming all given feature-defined resource demands have the same value). (5) If none of these cases applies or if no feature-defined resource demand is specified (i.e., an empty set), we assume the resource demand of the software component in context of the resource type to be nil. Generally, the mapping function should be adapted if further resource type properties or resource demand dependencies are added in the future.

Figure 4 visualises the mapping-function for the system response time in the running example. Both the features `navigation-system` and `real-time-traffic-system` demand an upper bound to the system's response time (50 ms and 30 ms). As the according resource type is non-additive and describes a lower bound, the mapping computes the minimum of both given values (i.e., 30 ms).

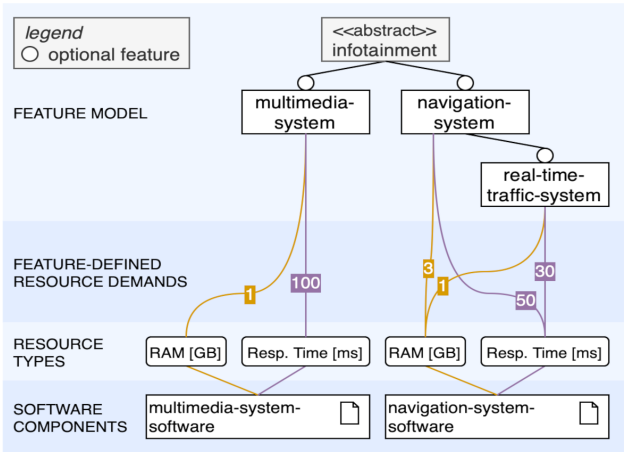

Figure 3: Feature-defined demands in the running example.

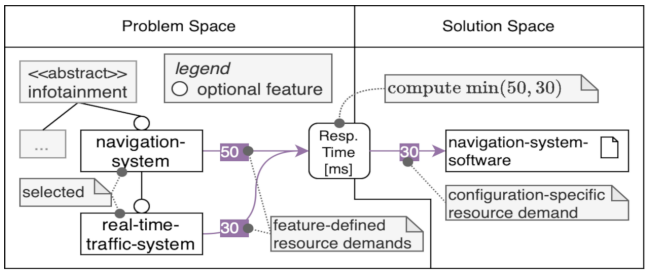

Figure 4: Mapping of feature-defined resource demands to configuration-specific resource demands.

4 Configuration Realisability & Update-Ability

In Section 3, we introduced a model of variability-aware resource demands, which can be mapped to concrete resource demands of software components for a particular configuration. In order to decide the realisability of this configuration, we must ensure that all resources demands of the software components are satisfied by provided resources of hardware components. Therefore, we formalise a CSP for resource allocation. The CSP can be computed by a COTS solver (e.g., *Z3 Theorem Prover*). We describe how the CSP can be instantiated to test the realisability and, afterwards, the update-ability of a configuration in the running example and define how the consistency of a product line can be derived with the proposed realisability analysis method.

4.1 Constraint Satisfaction Problem for Resource Allocation

A CSP consists of constants, variables and constraints. In the following, we introduce the necessary parts, assemble a global problem representation and give information about its computation.

Constants. We define all resource types as constants and use the three resource type properties for further case distinctions. We also define the values of all demanded resources of software components and all provided resources of hardware components as constants.

Variables. We define the assignment from a resource demand of a software component to a resource provisioning of a hardware component in context of a resource type with a boolean *resource assignment variable*. There exists one resource assignment variable from each software to each hardware component in context of each resource type. Analogous, we model the assignment of a software component to a hardware component with a boolean *component assignment variable*. There exists one component assignment variable from each software to each hardware component.

Assignment Constraints. We define constraints T_{asgn} for the relationship between variables for resource- and component-wise assignments. These constraints ensure that all resource demands of a software component are assigned to the resource provisionings of the same hardware component over all resource types, i.e., are not scattered over multiple hardware components because of their differing resource type. In the context of this work, we argue that the deployment of a software component should be restricted to a single hardware component, for example due to latency. Thus, a software component is assigned to a hardware component iff all resource demands of the software component are assigned to resource provisionings of the hardware component. We further define constraints T_{bnd} to ensure that each software component is assigned to *exactly* one hardware component.

Resource Comparison Constraints. To ensure that each demanded resource of a software component is actually satisfied, we must compare it to a provided resource of a hardware component. This comparison is done in context of a concrete resource type (i.e., only demanded and provided resources of the same resource type are compared) with its three properties additivity, exclusivity and boundary type. As two resource type properties are binary (additivity, exclusivity) and one is ternary (boundary type), there are exactly twelve distinct combinations of resource type properties possible. Instead of naively defining different constraints for each of these twelve characteristics, we state two logical equivalences so that only four different constraints are necessary to express all twelve characteristics (visualised in Figure 5):

(1) Providing resources in context of an exclusive resource type ensures that a maximum of one software component may be assigned to the hardware component. For hardware components in turn, this implies that the context of an exclusive resource type is, by definition, equivalent to the context of a non-exclusive resource type with an upper bound of one for the number of assigned software components.
(2) We defined the boundary types `LOWER` and `UPPER` to be non-strict (i.e., $\leq$ and $\geq$). For the context of a resource type, this implies, by definition of non-strict monotonicity, that ensuring the boundary type `EXACT` (i.e., $=$) is equal to ensuring both the boundary types `LOWER` and `UPPER`.

We define resource comparison constraints T_{comp} for the remaining four characteristics in context of a hardware component and a resource type as follows: (1) In context of a non-additive, non-exclusive, lower-bound resource type, we ensure that each demanded resource value does not exceed a provided resource value. (2) In context of a non-additive, non-exclusive, upper-bound resource type, we ensure that each provided resource value does not exceed a demanded resource value. (3) In context of an additive, non-exclusive, lower-bound resource type, we ensure that the sum of all demanded resource values do not exceed the provided resource value. (4) In context of an additive, non-exclusive, upper-bound resource type, we ensure that the provided resource value does not exceed the sum of all demanded resource values.

Assembling the global CSP. We assemble the global CSP T_{rls} to decide the realisability of a configuration as conjunction of three parts: (1) The conjuncted constraints T_{asgn} ensuring resource- and component-wise assignments, (2) the conjuncted constraints T_{bnd} ensuring assignment-bounds to hardware components and (3) the conjuncted resource comparison constraints T_{comp}.

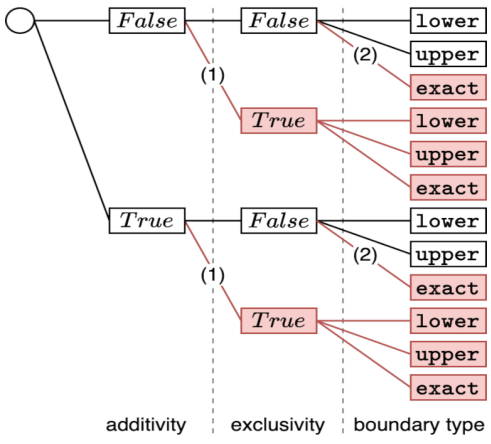

Figure 5: Four resource comparison constraints for twelve possible characteristics of resource type properties.

Computation & Implications on Solution. The CSP T_{rls} can be computed by COTS solvers (e.g., Z3). If the computation yields no solution (i.e., the CSP T_{rls} is not satisfiable), there exists at least one resource demand constraint which can not be satisifed by given resource provisionings. Therefore, we assume the configuration not to be realisable. If a solution exists (i.e., the CSP T_{rls} is satisfiable), we assume the configuration to be realisable.

If the CSP T_{rls} is realisable, we know which software component is deployed to which hardware component so that all resource demands are satisfied. The concrete assignments between the components are defined by the subset of satisfied component-assignment-variables. Although many solutions of the resource allocation problem may exist, it generally suffices to compute one possible solution to show the realisability of the configuration in question.

4.2 Realisability & Update-Ability in the Running Example

Realisability. As introduced earlier, Figure 3 shows the resource demands of each feature in the running example. Above that, the hardware comp `pu-1` provides 4 GB of RAM and a maximum system response time of 30 ms. However, the provided RAM of `pu-1` (4 GB) does not suffice if both the features `multimedia-system` (demands 1 GB) and `real-time-traffic-system` (demands 1 GB) are selected besides `navigation-system` (demands 3 GB). Therefore, a configuration is realisable as long as only one of the two features `multimedia-system` and `real-time-traffic-system` is selected (mutual exclusion). To fix the realisability issues, the manufacturer dimensions the hardware comp `pu-1` so that it now provides 5 GB of RAM. This implies that configurations with both the two features `multimedia-system` and `real-time-traffic-system` selected are now also realisable.

Update-Ability. After some time in production, the manufacturer wants to deploy an over-the-air update to the feature `navigation-system`, which offers a better route planning algorithm. With the update, this feature now demands 4 GB of RAM (instead of formerly 3 GB). If, additionally, both the features `multimedia-system` (demands 1 GB) and `real-time-traffic-system` (demands 1 GB) are selected besides `navigation-system` (demands 4 GB), the provided RAM of `pu-1` (provides 5 GB) does not suffice. Therefore, analogously to realisability, the update may only be deployed to products which are equipped with only one of the two features `multimedia-system` and `real-time-traffic-system`. In general, deciding the update-ability of products in field is only relevant to configuration variants which are equipped by the updated feature, because only these variants are affected by the update.

4.3 Deciding Consistency of Problem Space & Solution Space

In Section 4, we proposed a solution space analysis method to decide the realisability of a single configuration based on a resource allocation problem. By applying the analysis method to all possible configurations ($\mathcal{CS}$), we can theoretically construct the set of all realisable configurations ($\mathcal{RS}$). In the problem space, by deciding the validity of each possible configuration, we can construct the set of all valid configurations ($\mathcal{VS}$) [2]. By comparing both the set of valid and the set of realisable configurations ($\mathcal{VS}$ and $\mathcal{RS}$), we can identify mismatches (inconsistencies) between the problem and solution space. However, this is infeasible in practice due to exponential complexity with respect to the number of features ($\mathcal{O}(2^{|F|})$) [16] and to NP-completeness of the realisability analysis method ($\mathcal{O}(2^{|\text{CSP-variables}|})$) [6].

To enable practical applicability, we adapt the definition of consistency by only considering *valid* configurations for realisability analysis. We redefine problem and solution space as consistent iff all valid configurations are also realisable. Furthermore, this definition implies that we can show the in-consistency of a product line if we find one valid configuration that is not realisable. This re-definition is useful especially for larger product lines, such as in the automotive domain, and where realisability analysis is applied to valid configurations only.

5 Evaluation

We evaluate our modelling approach for variability-aware resource demands (Section 3) and our solution space analysis method for realisability (Section 4) and consistency (Section 4.3) for two different aspects: feasibility and performance.

5.1 Research Questions

With the evaluation results, we aim to answer the following research questions in terms of feasibility and performance:

RQ1 [Feasibility]: How can an existing feature model be extended by resource specifications and solution space analysis? To investigate the feasibility of our modelling approach and our realisability analysis method, we apply our approach to the BCS case study. We extend the BCS feature model by software and hardware components as well as resource types, resource provisionings and variability-aware resource demands. We compute the realisability of each valid configuration of the feature model and compare the results to a derived ground truth. Finally, we discuss the implications of the realisability analysis on the consistency of the constructed product line.

RQ2 [Performance]: How does the resource and consistency analysis method perform regarding run-time? In Section 4.3, we proposed a definition of consistency between the problem and solution space of a product line. Proving consistency for a concrete product line along this definition consists of multiple, computationally complex steps. First, the number of possible configurations to investigate is exponentially large with respect to the number of features in the feature model [16]. Second, the computation of the resource allocation problem is generally considered to be NP-complete with respect to the number of assignment variables involved [6]. Third, both steps mentioned before are nested as we have to decide the realisability for each valid configuration in order to decide the consistency of the product line. Hence, we evaluate the run-time of deciding the consistency of the extended BCS case study.

5.2 Setup & Methodology

Subject System. The problem space of the BCS case study [20] consists of a feature model with 28 features. It can be divided into three sub-trees: (1) The sub-tree `hmi` concerns a user interface. (2) The sub-tree `doors` concerns convenience and safety functionality

such as electrical and heatable exterior mirrors. (3) The sub-tree `security` concerns functionality such as an alarm system and central locking. The feature model yields $|\mathcal{CS}_{\text{BCS}}| = 2^{28} \approx 268 \cdot 10^6$ possible feature combinations with $|\mathcal{VS}_{\text{BCS}}| = 11616$ of them valid. In the feasibility evaluation (RQ1), we extend the BCS case study by solution space artefacts, such as components, resources and -types.

Ground Truth for Configuration Realisability. To keep track of the realisability of configurations, we model the feature-defined resource demand so that certain configurations are not realisable (i.e., yield incompatible solution space artefacts). In detail, we model a configuration as realisable iff it does not contain the feature `alarm--system-interior`. This feature leads to a demanded number of security video cameras of five, which can not be satisfied by any of the specified hardware components.

Software Tool Support. To handle feature models and problem space analysis, we use *FeatureIDE* in version `3.11.1` installed in *Eclipse Modelling Tools 2023-12*[1], version `4.30.0`. Our proposed analysis method to decide the realisability of configurations is implemented in the Python programming language in version `3.9.6`. To solve CSPs, we use the *Z3 Theorem Prover* [4] implemented as Python package `z3-solver`[2] in version `4.12.6.0`.

Z3 Theorem Prover Parameters. For analysing the realisability of a configuration, we solve the according CSP without any parameter tweaking (i.e., as parameterised by default[3]). For the performance evaluation, we randomise the computation of valid configurations by setting a random seed and randomise solver decisions[4].

Hardware Device. We execute all mentioned tools on an Apple MacBook Pro 2023 (`Mac14,9`) with Apple Silicon M2 Pro CPU.

Run-Time Evaluation Procedure. Listing 1 shows the procedure of our run-time evaluation (RQ2), where we measure three datapoints: (1) The total time to decide the consistency of the product line and, within the total time, (2) the accumulated time to generate valid configurations and (3) the accumulated time to decide realisabilities of the configurations. As the inconsistency of a product line is considered given when the first non-realisable configuration is found, the run time of the decision procedure highly depends on the order of configurations investigated. To minimise this influence, we repeat the decision procedure 1000 times with random configuration sequences. Finally, we discuss the results and highlight important findings in the computation procedure.

5.3 Results

In the following, we present the results of our evaluation. In context of feasibility, we extended the the BCS product line by solution space artefacts and analysed the consistency of the extended product line. In context of performance, we investigated the run-time necessary to decide the product line's consistency. The full evaluation data and code can be found online [22].

[1] https://www.featureide.de/ and https://www.eclipse.org/
[2] https://github.com/Z3Prover/z3 and https://pypi.org/project/z3-solver/
[3] https://microsoft.github.io/z3guide/programming/Parameters
[4] parameter `smt.random_seed=true`, `smt.phase_selection=5`, `sat.phase=random`

Listing 1: Prodecure to measure the run-time of deciding a product line's consistency (RQ2).

```
repeat consistency decision 1000 times:
    start(total_time)
    FM, FD, RP, RT ← load from files
    seed ← draw fixed seed randomly
    start(val_time)
    C ← draw random valid configuration with seed
    end(val_time) and add to accumulator
    while (C is given):
        map resource demands for configuration C
        construct CSP T_rls for configuration C
        start(rls_time)
        realisable ← analyse realisability of config. C by solving T_rls
        end(rls_time) and add to accumulator
        if (not realisable):
            finish consistency decision (break)
        else:
            start(val_time)
            C = draw next random valid configuration with seed
            end(val_time) and add to accumulator
    end(total_time)
    resume (total_time, accumulated val_time, accumulated rls_time)
```

RQ1: Solution Space Artefacts & Analysis. In the solution space, we extend the feature model of the BCS case study with software components, hardware components and resource types. We specify four software components: The software component `hmi-software` encapsulates the implementation of the user interface. The software components `power-window-control-software` and `exterior-mirror-control-software` address the operation of power windows and exterior mirrors. The software component `security-software` controls security installations, such as the alarm system and the central locking. Furthermore, we specify two hardware components: The hardware component `security-hardware` provides exclusive resources for security installations of the car. The hardware component `infotainment-hardware` is a central processing unit for all non-critical tasks in the portrayed system. Table 2 shows an excerpt of in total eight specified resource types: We define a resource type to model power consumption constraints on the equipped parts, a resource type for time-constraints on an automatic re-lock of the car's doors and a resource type to ensure that an exact number of security video cameras are given.

We specify 27 variability-aware resource demands of features. Figure 6 shows an excerpt of the BCS feature model with attributed features, resources types and software components. The features `exterior-mirror-electric` and `exterior-mirror-heatable` specify a power consumption of 1 W and 20 W in context of software component `exterior-mirror-control-software`. The features `alarm-system` and `alarm-system-interior` specify a number of security video cameras of four and one in context of software component `security-software`. In context of the same software component, the feature `central-locking-automatic` specifies an automatic re-lock time of 1 min.

By applying our proposed method to analyse the realisability of each valid configuration, we identified 6528 configurations to be realisable ($|\mathcal{RS}_{\text{BCS}}| = 6528$), thus 5088 configurations not realisable.

ID(k)	Description	Additivity	Exclusivity	Boundary T.	Unit
1	Power Specification	**True**	**False**	LOWER	W
4	Automatic Relock Time	**False**	**True**	UPPER	min
5	no. Security Video Cam.	**True**	**True**	EXACT	

Table 2: Excerpt of resource types in the ext. BCS case study.

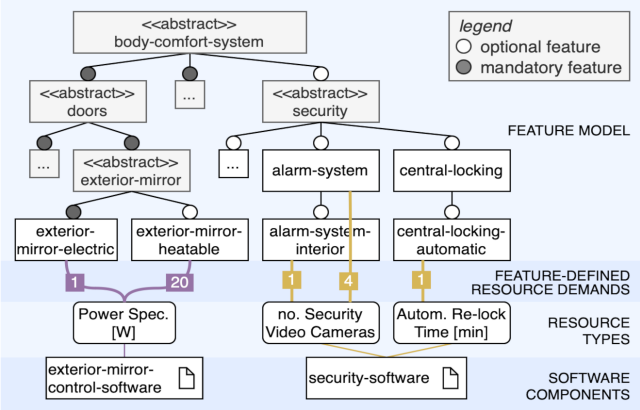

Figure 6: Excerpt of feature-defined resource demands in the extended BCS case study.

RQ2: Performance of Consistency Decision. We measured the total run-time as well as the partial run-times to compute valid configurations and the configuration's realisabilities of analysing the consistency of the BCS product line. Figure 7a shows a combined violin-box-plot-diagram of the three time-measurements distributed over 1000 repetitions. The measured data is given on a logarithmic scale in seconds. The partial run-time to compute valid configurations was ~ 10.36 ms in median and the partial run-time to compute the configuration's realisabilities was ~ 13.97 ms in median. The total run-time for deciding the consistency of the product line was ~ 206.23 ms in median. All three time-measurements have modes below its medians and sparse occurrences of higher run-times. The comparison of the partial run-times and the total run-time shows a computation overhead of ~ 181.9 ms between the medians.

Figure 7b shows the number of configurations which were visited in order to decide the consistency of the BCS product line, distributed over 1000 repetitions: 3.0 configurations were necessary in median until a consistency-breaking configuration was found. Similar to the run-time measurements, the distribution has a mode below its median. In worst-case, 39 configurations were visited until a consistency-breaking configuration was found.

5.4 Discussion

RQ1 [Feasibility]: How can an existing feature model be extended by resource specifications and solution space analysis? For realisability analysis, we derive the ground truth for each configuration along the presence of the feature `alarm-system-interior`. There were 6528 configurations identified as realisable and 5088 configurations as non-realisable. We examine that in every non-realisable configuration, the feature `alarm-system-interior` is selected and, vice versa, not selected in every realisable configuration. As these results are equivalent to the ground truth, our method to decide the realisability of configurations is evaluated positively in context of the BCS product line. As we found *all* expected non-realisable configurations, we imply that our method also decides the consistency between problem and solution space of the BCS product line.

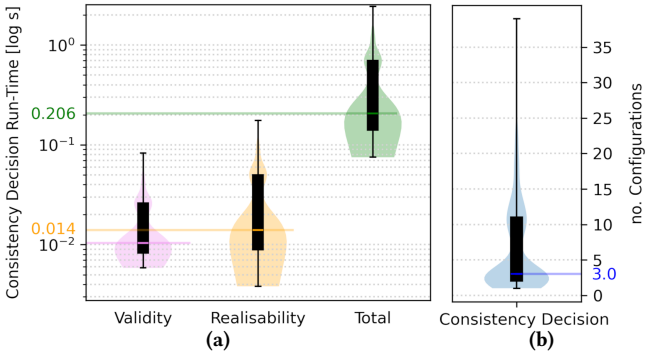

Figure 7: (a) Run-time and (b) number of visited configurations until the extended BCS product line is decided.

RQ2 [Performance]: How does the resource and consistency analysis method perform regarding run-time? Although the process to decide the consistency of a product line consists of multiple, computationally complex steps, it is computed in ~ 206.23 ms in median for the BCS product line, as shown in Figure 7a. Additionally, the time-measurements show that only the minority of the total run-time (approximately 12 %) is spent on actually solving CSPs. We account the run-time overhead of ~ 181.9 ms to computation steps such as loading file content from the hard drive, constructing data structures as well as constructing the CSPs itself.

Furthermore, Figure 7b shows that only 3.0 configurations in median were analysed until a valid configuration was found which is not realisable. We modelled resource demands in the BCS product line such that a configuration is realisable iff it does not contain the feature `alarm-system-interior`. As this feature is optional, one half of all valid configurations contains this feature (not considering cross-tree constraints). Thus, we expect a chance of 50 % of drawing a random valid configuration with this feature selected. This expectation is supported by comparing the number of realisable to non-realisable configurations of 6528 and 5088 respectively.

In conclusion, for deciding a product line's consistency, we argue that the number of executed steps is below its worst-case ($O(2^{|F|+|\text{CSP-variables}|})$) in practice, because we expect to find an existing consistency-breaking configuration early in the procedure.

5.5 Limitations & Extensions

We understand our approach as a baseline for resource-based realisation analysis of cyber-physical systems. Thus, our work serves as a starting point for further research. In our evaluation, we identified two limitations in terms of practicality and scalability and propose the following advancements: (1) We envisage an expert-driven case study on the practicality of our approach to gain experience in designing software and hardware components as well as collect data for the specification of resources and types. (2) We aim to evaluate

the scalability of our realisability analysis method applied to larger systems, where we expect the run-times to grow exponentially with the number of features in a feature model and the number of solution space artefacts.

To overcome these limitations, we have identified three extensions to our realisability analysis method as follows: (1) Our current procedure is product-based [26], as we analyse the realisability for each configuration separately. As mentioned above, this could lead to scalability issues for larger systems. Therefore, we focus on more scalable strategies such as family-based procedures [26]. (2) We aim to investigate the design of more complex systems, for example with support for presence conditions on features and realisation artefacts, as this would increase the generality of our approach beyond plain feature-to-artefact mappings. (3) Our resource allocation modelling is of static nature, as we assume the presence of *all* software components and their resource demands *simultaneously*. In practice, software functionality is often used dynamically or even with mutual exclusion. For example, in a car infotainment, a music streaming application may not be displayed simultaneously to the navigation. Hence, the hardware resources are reallocated and the system design therefor may be optimised for cost-effectiveness. Supporting dynamic reallocation and optimisation objectives would advance our analysis method towards broader applicability.

5.6 Threats to Validity

Internal. We identified two major internal threats to validity concerning our performance evaluation: (1) The generation of configurations with the Z3 Theorem Prover is not uniform-randomly distributed over the sample space, hence leading to more configurations necessary to decide a product line's consistency. We argue that, with uniform-randomly sampled configuration sequences, there are even lower numbers of configurations, and thus less time necessary to decide the consistency of a product line. (2) We use random seeds for the generation of configurations with the Z3 Theorem Prover. However, different random seeds do not imply different configuration sequences. Thus, we ensured considering only distinct sequences in the time-measurements and repeated them 1000 times to mitigate influences of single configuration sequences.

External. As mentioned in the limitations, our evaluation with the extended BCS case study only is a major external threat to validity and generality of our approach for arbitrary product lines regarding scalability. As our proposed method is product-based [26], we expect the run-times of our realisability analysis method to grow exponentially with respect to the number of features in a feature model *and* the number of solution space artefacts. Furthermore, we expect a worse performance of our consistency analysis method with lower chance of drawing a consistency-breaking configuration early (e.g., compared to the BCS case study with aproximately 50 % chance). This directly influences the number of configurations, thus the run-time necessary to decide the product line's consistency.

6 Related Work

Our proposed method to analyse the realisability of configurations is related to existing work in context of problem space, solution space and consistency analysis as well as resource allocation analysis and product line modelling.

Problem Space Analysis. Existing approaches to problem space analysis [10, 26] typically concern the feature model of a product line and often are contextualised by software product lines. Our approach combines existing problem space analysis with a novel solution space analysis in generic cyber-physical product lines.

Solution Space Analysis. Hentze et al. [14] present a solution space analysis method by integrating solution space constraints (e.g., dependencies between solution space artefacts) into the problem space of the product line, thus enabling problem space analysis techniques [10] to be applied to the whole product line. However, this approach is of limited artefact resolution as only *components* are considered, while our approach considers finer-grained *resources*.

Kästner et al. [17] and Gazillo et al. [11] present static code checking analysis for variability-aware source code artefacts of software product lines. Moreover, Kim et al. [15] and Shi et al. [24] present tools to identify code similarities in software product lines with product-based, static software analysis [15] and feature interaction analysis [24]. Each approach is described for software product lines only, while our solution space analysis method concerns the interplay between generic software and hardware components.

Inconsistency Analysis. Thüm et al. [25] and El-Sharkawy et al. [9] reason about inconsistencies between problem and solution space caused by feature model design. Thüm et al. identify abstract features (without solution space artefacts) as cause of mismatch between modelled and actual variability, while El-Sharkawy et al. analyse configuration mismatches in the linux kernel. Hentze et al. [14] identify inconsistencies by comparing the problem space feature model to one which also respects solution space constraints. Le at al. [18] present a concept for reverse-engineering and comparing a feature model from and to its source code. Our approach contributes a further inconsistency analysis method by comparing the number of modelled to the number of realisable configurations.

Product-Line- & Automotive-Modelling. Ananieva et al. [1] present a meta-model of a product line expressing a feature model (problem space) and realisation fragments (solution space). Their work is extended by Wittler et al. [28] who modelled the solution space with components, resources and resource types. In the automotive industry, several modelling approaches exist, for example SysML [21] and EAST-ADL [8], where a system is represented on different abstraction levels with different engineering models. Our work contributes an analysis for realisability of configurations and consistency of product lines based on the model of Wittler et al. [28].

Product Lines & Resource Allocation. Leite et al. [19] and White et al. [27] both combine product line modelling with resource allocation in the solution space. Leite et al. [19] use feature models to model resource types and concrete configurations to express a hardware's resource provisionings. With specified resource demands, an assignment to a hardware is computed so that an optimisation criteria (e.g., cost) is minimised. In turn, White et al. [27] compute system configurations which optimise resource usage, e.g., maximise the usage of provided resources on given hardware and keeping costs low for the system's demanded resources. Both approaches express resources types with product lines while our approach separates product variability from solution space design.

7 Conclusion & Future Work

In this paper, we proposed a novel baseline of solution space analysis for the realisability of products in configurable cyber-physical systems, such as in automotive systems. We gave a description of realisation artefacts and specified how variability-aware resource demands can be mapped to configuration-specific resource demands for a single product. We defined a CSP based on demanded and provided resources in the product and gave implications of the realisability of configurations on the consistency of a product line. In the evaluation, we investigated the feasibility and performance of our method, based on the automotive BCS case study [20]. We showed that our analysis method can identify all non-realisable configurations in the modelled subject system and can decide the product line's consistency by finding a non-realisable configuration in a median time of 206 ms, where a major overhead of the time is spent on file reading and data structuring. In the future, we aim to overcome limitations of our ongoing work in terms of applicability and scalability by gaining practical experience in system design and focusing on family-based approaches.

Acknowledgments

This work was partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263, by the German Federal Ministry for Economic Affairs and Climate Action – "Software-Defined Car" (SofDCar) – 19S21002 and by the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg – "Innovations Campus Future Mobility" (ICM).

References

[1] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A conceptual model for unifying variability in space and time. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. ACM, Montreal Quebec Canada, 1–12. https://doi.org/10.1145/3382025.3414955

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37521-7

[3] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Active Flow and Combustion Control 2018*, Rudibert King (Ed.). Vol. 141. Springer International Publishing, Cham, 491–503. https://doi.org/10.1007/11431855_34 Series Title: Notes on Numerical Fluid Mechanics and Multidisciplinary Design.

[4] Nikolaj Bjørner. 2012. Taking Satisfiability to the Next Level with Z3. In *Automated Reasoning*, Bernhard Gramlich, Dale Miller, and Uli Sattler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–8.

[5] Paul Clements and Linda Northrop. 2002. *Software product lines: practices and patterns*. Addison-Wesley, Boston.

[6] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/800157.805047 event-place: Shaker Heights, Ohio, USA.

[7] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. https://doi.org/10.1145/1995376.1995394 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[8] EAST-ADL Association. 2024. *Electronics Architecture and Software Technology - Architecture Description Language (EAST-ADL)*. Specification Language. http://east-adl.info/2.2.0 Accessed 2024-08-16.

[9] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A (SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3106195.3106208 event-place: Sevilla, Spain.

[10] José A. Galindo, David Benavides, Pablo Trinidad, Antonio-Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (May 2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[11] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. *SIGPLAN Not.* 47, 6 (June 2012), 323–334. https://doi.org/10.1145/2345156.2254103 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[12] Weiwei Gong and Xu Zhou. 2017. A survey of SAT solver. Rome, Italy, 020059. https://doi.org/10.1063/1.4981999

[13] 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 2022)*. Association for Computing Machinery, New York, NY, USA, 135–147. https://doi.org/10.1145/3564719.3568695 event-place: Auckland, New Zealand.

[14] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2022. Quantifying the Variability Mismatch between Problem and Solution Space. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3550355.3552411 event-place: Montreal, Quebec, Canada.

[15] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development (AOSD '11)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/1960275.1960284 event-place: Porto de Galinhas, Brazil.

[16] Charles W. Krueger. 2006. New methods in software product line practice. *Commun. ACM* 49, 12 (Dec. 2006), 37–40. https://doi.org/10.1145/1183236.1183262 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[17] 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 *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, New York, NY, USA, 805–824. https://doi.org/10.1145/2048066.2048128

[18] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating Consistency between a Feature Model and Its Implementation. In *Safe and Secure Software Reuse*, John Favaro and Maurizio Morisio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16. https://link.springer.com/chapter/10.1007/978-3-642-38977-1_1

[19] Alessandro Ferreira Leite, Vander Alves, Genaina Nunes Rodrigues, Claude Tadonki, Christine Eisenbeis, and Alba Cristina Magalhaes Alves de Melo. 2015. Automating Resource Selection and Configuration in Inter-clouds through a Software Product Line Method. In *2015 IEEE 8th International Conference on Cloud Computing*. 726–733. https://doi.org/10.1109/CLOUD.2015.101

[20] Sasha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. 2012. *Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study*. Technical Report 2012-07. Technische Universität Braunschweig. https://www.isf.cs.tu-bs.de/cms/team/lity/bcs_tubs_tech_rep_V1_4.pdf

[21] Object Management Group (OMG). 2024. *System Modeling Language (SysML)*. Specification Language. https://www.omg.org/spec/SysML Accessed 2024-08-16.

[22] Philip Ochs. 2024. KIT-TVA/solution-space-realisation-checker: mase-2024. https://doi.org/10.5281/ZENODO.13310338

[23] Stuart J. Russell and Peter Norvig. 2022. *Artificial intelligence: a modern approach* (fourth edition. global edition ed.). Pearson, Harlow, United Kingdom. OCLC: 1242911311.

[24] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Fundamental Approaches to Software Engineering*. Vol. 7212. Springer Berlin Heidelberg, Berlin, Heidelberg, 270–284. https://doi.org/10.1007/978-3-642-28872-2_19 Series Title: Lecture Notes in Computer Science.

[25] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *2011 15th International Software Product Line Conference*. IEEE, Munich, Germany, 191–200. https://doi.org/10.1109/SPLC.2011.53

[26] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (June 2014). https://doi.org/10.1145/2580950 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[27] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. 2007. Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference (SPLC 2007)*. 129–140. https://doi.org/10.1109/SPLINE.2007.19

[28] Jan Willem Wittler, Thomas Kühn, and Ralf Reussner. 2022. Towards an Integrated Approach for Managing the Variability and Evolution of Both Software and Hardware Components. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B (SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 94–98. https://doi.org/10.1145/3503229.3547059 event-place: Graz, Austria.