Karlsruhe Institute of Technology

Test, Validation, Analysis

Combined Modeling of Software and Hardware with Versions and Variants

Master's Thesis
of

Philip Ochs

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Second Reviewer:	Prof. Dr. Ralf Reussner
Advisors:	M.Sc. Tobias Pett
	M.Sc. Jan Willem Wittler

Completion period: 06. February 2023 – 07. August 2023

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 07.08.2023

Zusammenfassung

Produktlinien ermöglichen die Entwicklung variabler, konfigurierbarer Produkte auf Basis von Konfigurationen. Eine valide Konfiguration wird im Problemraum einer Produktlinie abgeleitet, während ihre Realisierbarkeit zu einem Produkt im Lösungsraum entschieden wird. Eine valide, aber nicht realisierbare Konfiguration führt zu einer Inkonsistenz zwischen Problem- und Lösungsraum, da die Menge der tatsächlich baubaren Produkte im Lösungsraum nicht mit der definierten Variabilität im Problemraum übereinstimmt. Für große Produktlinien ist es händisch schwierig zu entscheiden, ob Inkonsistenzen existieren, da die Anzahl der Konfigurationen in Produktlinien exponentiell wächst. In dieser Arbeit wird die Konsistenz von Problem- und Lösungsraum adressiert und zwei formale Methoden auf Basis von Bedingungserfüllungsproblemen vorgeschlagen, um die Menge der Konfigurationen zu berechnen, die sowohl valide als auch realisierbar sind. Die *Sampling-Methode* entscheidet die Realisierbarkeit für jede Konfiguration aus einer Menge von validen Konfigurationen einzeln. Damit kann im Voraus überprüft werden, welche bestehenden Konfigurationen auch nach einer Softwareaktualisierung noch funktionsfähig sind. Die *kombinierte Methode* berechnet direkt Konfigurationen, die sowohl valide als auch realisierbar sind, für weitere Produkttests. Beide Methoden werden anhand zweier Fallstudien mit exemplarischen Produktlinien evaluiert. In den Ergebnissen wird gezeigt, dass beide Methoden alle Inkonsistenzen zwischen Problem- und Lösungsraum in den Produktlinienbeispielen identifizieren können.

Abstract

Product lines allow to derive variable, customisable products based on configurations. While a valid configuration is derived in the problem space of a product line, its realisability to a product is decided in the solution space. A valid but non-realisable configuration leads to an inconsistency between problem and solution space, because the set of actually derivable products in the solution space does not match the defined variability in the problem space. For large product lines, the manual decision about this consistency is infeasible because the number of configurations grows exponentially. In this thesis, we address this decision and propose two formal methods to compute the set of configurations which are both valid and realisable, based on constraint satisfaction problems. The *Sampling Method* decides the realisability for each configuration in a sample of valid configurations. This allows to check in advance whether an existing configuration will still work after receiving a software update. The *Combined Method* computes configurations both valid and realisable directly for further product testing. We evaluate both methods in two case studies with exemplary product lines. The results show that both methods can identify all mismatches between the problem and the solution space in the applied product line examples.

Contents

List of Figures

List of Tables

Acronyms

ALU	Arithmetic Logic Unit
API	Application Programming Interface
BCS	Body Comfort System
COTS	Commercial Off-The-Shelf
CPS	Cyber-Physical System
CS1	Case Study 1
CSP	Constraint Satisfaction Problem
CSV	Comma-Separated Values
CU	Computation Unit
FM	Feature Model
GT	Ground Truth
IDE	Integrated Development Environment
PL	Product Line
RAM	Random Access Memory
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
SPLE	Software Product Line Engineering
UCM	Unified Conceptual Model
UML	Unified Modeling Language
XML	Extensible Markup Language

1. Introduction

Industry products increasingly combine physical and digital parts, such as hardware and software components. The interconnection and the tight coupling of these components describe a Cyber-Physical Systems (CPSs) [Nat]. These systems are becoming more and more ubiquitous in everyday life. Manufacturers have to face new challenges in order to develop and produce CPS, for example different life cycles of hardware and software components. For products in development and production, hardware components can be considered static with fixed, immutable functionality. Software components, on the other hand, are in a dynamic environment where rapid changes in functionality are possible. Even later, for products in use, technologies such as over-the-air update and functions on demand enable short release cycles and fast deployment for software components.

The properties of CPSs enable manufacturers to introduce variability to their products. Certain parts of the system can be replaced while a set of core building blocks remain the same. In the automotive industry, for example, a customer can configure a car to be equipped either with a small radio or with a navigation system. In addition to production (build-time variability), run-time variability refers to variability of already existing products, for example the deployment of a revised route planning algorithm to the navigation system. Offering variability as a more and more common pattern to customers is called mass customisation and can be managed in a Product Line (PL) [JTDL98].

A PL, in the scope of this thesis, consists of a problem space and a solution space [CN02]. Theoretical aspects of variability are managed by the Feature Model (FM) in the problem space. A user-experienceable functionality is called a feature and addresses a single Variation Point. Both the small radio and the navigation system are features addressing the decision about the infotainment system of the car. Features may also have dependencies on other features, for example, a navigation system may require access to the internet. As a PL grows in size and complexity (e.g. the number of features increases), the number of configurable, distinct products grows combinatorically. Hence, analysing and managing the PL by hand becomes difficult. Several tools have been proposed for these tasks, for example the FeatureIDE [TAK+14].

In the solution space, the realisation of configured products is considered. The core building blocks of the PL as well as the selected features are mapped to a corresponding set of solution space artefacts. In context of CPSs, these solution space artefacts are a heterogeneous set of hardware and software components. For instance, for a car which is equipped with a navigation system, these artefacts might be a GPS antenna and a navigation software.

In practice, it can be shown that configurations which are valid in the problem space are not necessarily realisable in the solution space [HSTS22]. Such configurations would yield a non-functioning product. Among other things, this can be caused by incompatible solution space artefacts like hardware and software components. For instance, a car equipped with a lightweight Computation Unit (CU) may not be able to satisfy the extensive resource demands of the navigation software.

More generally, once a valid but non-realisable configuration exists, the problem and solution space of the PL became inconsistent [HSTS22]. This mismatch means that a single configuration must not only be validated in the context of the FM, but also tested for realisability in the solution space. As it is generally computationally difficult to analyse a FM, it is even harder to decide on the consistency of the problem and solution space.

Goal of this Thesis

The production of variable products in CPSs raises new challenges for manufacturers, such as different life cycles of software and hardware artefacts. A valid configuration of a PL must not necessarily be realisable in the solution space, because extensive resource demands of updated software artefacts are not satisfied by old hardware artefacts anymore. This leads to a diverging problem and solution space, where realisable configurations can not be found by problem space analysis alone.

This thesis addresses the described challenge of keeping the problem and solution space consistent. As a prerequisite, we define a unified meta-model of hardware and software artefacts that can express actual solution space artefacts of a configuration. It must also support different life cycles of and dependencies between solution space artefacts. Furthermore, we propose a procedure to test configurations on their realisability and imply statements about the consistency of problem and solution space by analysing the compatibility of hardware and software artefacts. To accomplish these objectives, the thesis consists of three research questions as follows:

RQ1 How can solution space artifacts be formally described in a meta-model?

RQ2 How can the realisability of a valid configuration be decided?

RQ3 How can run-time variability be described?

Structure of the Thesis

The remainder of this document is structured as follows: In Chapter 2 a detailed, more technical explanation of the problem is given and a running example is introduced. Chapter 3 describes basic knowledge and defines relevant terms. The solution approaches of the thesis are introduced along the three research questions

in Chapter 4. A description of the implementation artefacts for tool support is given in Chapter 5. The contributions are evaluated, delimited and discussed in Chapter 6. An overview of related work is given in Chapter 7. A conclusion and outlook summarises the results of this thesis in Chapter 8.

2. Problem Statement

As PLs grow in the number of available features, the number of valid configurations can become combinatorically high. Yet, Manufacturers still have to analyse and test the products they offer to the customers. For cost and feasibility reasons, these steps have to be done in theory or on a representative subset of products, as not every possible product can be build beforehand [HPS+22]. Regarding a FM in the problem space of the PL, a variety of analysis methods have been proposed in the past [TAK+14]. On the other hand, meta-models and analysis methods of the solution space are less common, because the solution space of CPSs unifies both software and hardware engineering concepts with separated concerns [BZ18].

To give an example for the problem, we assume the following setting: A car is manufactured as a PL and can therefore be configured in the problem space using the FM shown in Figure 2.1. The customer can choose between two kinds of infotainment system: A small radio (feature `radio`) or a navigation system (feature `navigation-system`). The latter can optionally be enhanced by real-time traffic information with the feature `real-time-traffic-system`.

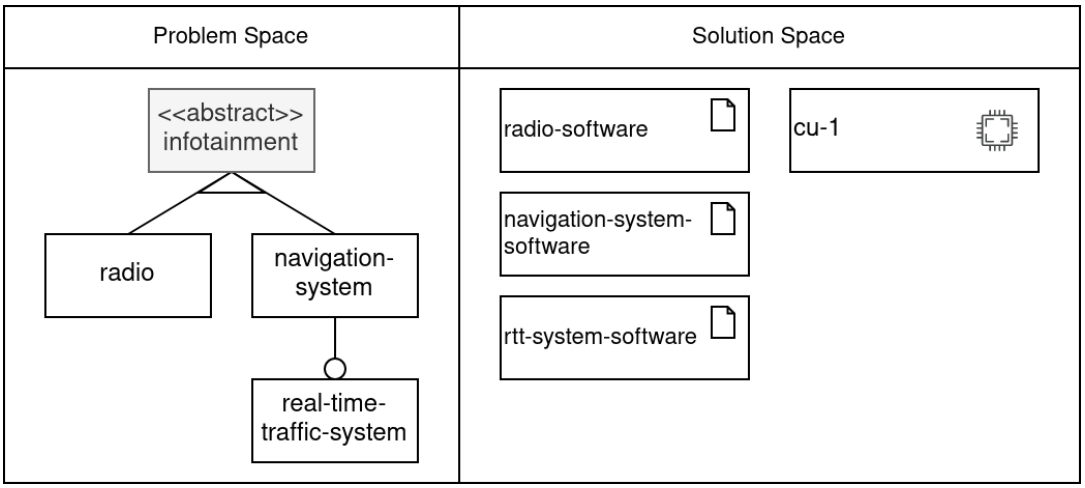

Figure 2.1: Illustrated problem and solution space for the PL of the running example

In the solution space, software artefacts for the features `radio` (`radio-software`), `navigation-system` (`navigation-system-software`) and `real-time-traffic-system` (`rtt-system-software`) are deployable to the hardware artefact `cu-1` (Computation Unit). Hardware artefacts are physical building parts of the car, for example semiconductor chips, and assumed as immutable.

In practice, for instance, a customer configures a car (as configuration $\mathcal{C}_A$) with the feature `radio`. Another customer configures a car (as configuration $\mathcal{C}_B$) with the feature `navigation-system`. A third customer also selects the feature `navigation-system` but with the enhancement of the feature `real-time-traffic-system` (configuration $\mathcal{C}_C$).

For configuration $\mathcal{C}_A$, the software artefact `radio-software` needs to be deployed to the hardware artefact `cu-1`. As shown in Figure 2.2, the software artefact `radio-software` demands at least 1 GB of Random Access Memory (RAM) in order to work properly. As the hardware artefact `cu-1` provides 4 GB of RAM in total, configuration $\mathcal{C}_A$ can be assumed as realizable.

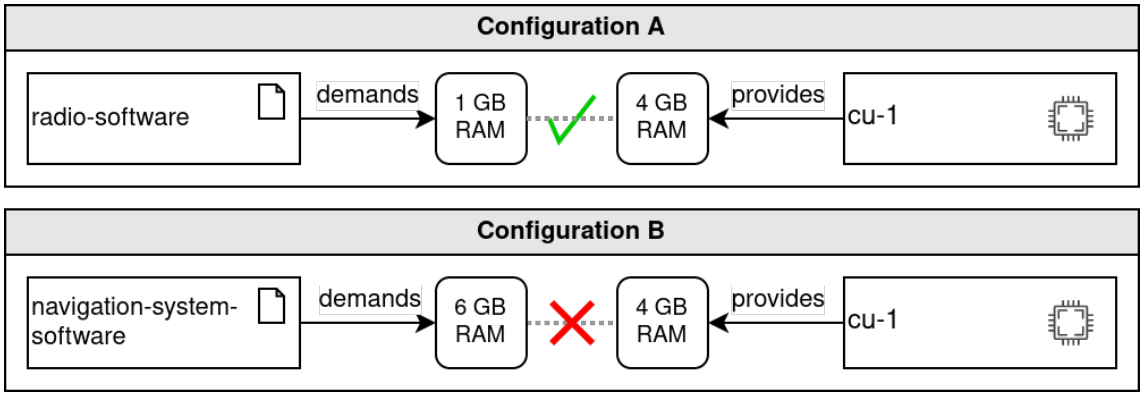

Figure 2.2: Configuration $\mathcal{C}_A$ of the running example is realisable, while configuration $\mathcal{C}_B$ (and implicitly $\mathcal{C}_C$) is not.

For configuration $\mathcal{C}_B$, the software artefact `navigation-system-software` is intended to be deployed to the hardware artefact `cu-1`. While the hardware artefacts remain the same, the software artefact `navigation-system-software` demands at least 6 GB of RAM. As this demand is not satisfied by the hardware artefact `cu-1`, configuration $\mathcal{C}_B$ can be assumed as non-realizable. As configuration $\mathcal{C}_C$ has at least the demands of configuration $\mathcal{C}_B$, it is also non-realizable. Although the realisability of the configurations $\mathcal{C}_A$, $\mathcal{C}_B$ and $\mathcal{C}_C$ can be decided manually in this example, this is not feasible for larger PLs with many configurations. Therefore, automatic decision methods are necessary (problem P1).

To demonstrate another problem, the following alteration of solution space artefacts is assumed: Besides the hardware artefact `cu-1`, the additional hardware artefact `cu-2` is also built into the car. It provides 8 GB of RAM and is therefore able to satisfy the demands of the software artefact `navigation-system-software` (at least 6 GB of RAM). The configuration $\mathcal{C}_B$ can now be realized. For configuration $\mathcal{C}_C$, the software artefact `rtt-system-software` additionally demands 2 GB of RAM. As hardware artefact `cu-2` has 2 GB of RAM left after the deployment of the software

artefact `navigation-system-software`, the software artefact `rtt-system-software` can be deployed to this hardware artefact. It can alternatively deployed to the hardware artefact `cu-1`, as this hardware artefact provides 4 GB of RAM. For many software and hardware artefacts with possibly nearly infinite different demands, the decision which software artefact is deployed to which hardware artefact is unknown and computationally hard to solve (problem P2).

Now we assume that an update for the feature `navigation-system` is released. The update increases the demands of the software artefact `navigation-system-software` from formerly 6 GB of RAM to 7 GB of RAM. The manufacturer wants to offer the update to customers with products already in the field and therefore needs to know, which existing cars are able to support the update. Figure 2.3 shows that every car can receive the update, but the ones with the optional feature `real-time-traffic-system` then definitely have to deploy the corresponding software artefact `rtt-system-software` to the hardware artefact `cu-1`. Generally, with an updated FM it is unknown which existing products would still be functioning and which changes in solution space artefact assignments have to be made concretely (problem P3).

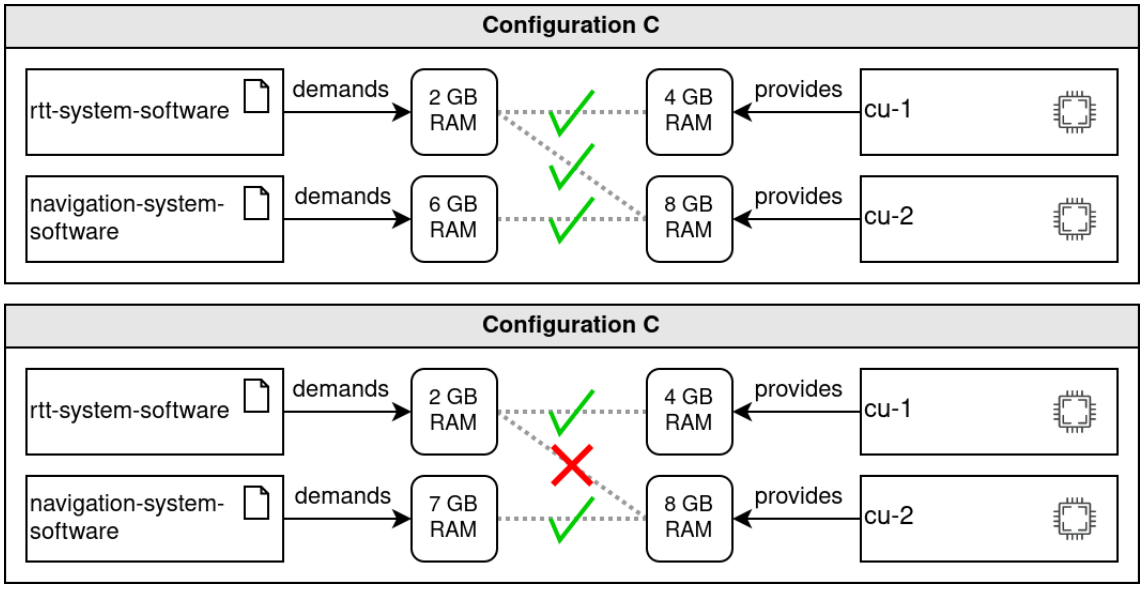

Figure 2.3: Before the update (top), software artefact `rtt-system-software` could be deployed to either hardware artefact `cu-1` or `cu-2`. After the update (bottom), the deployment to `cu-2` is no longer possible.

The research questions RQ1, RQ2 and RQ3 stated in Chapter 1 address the problems P1, P2 and P3 by providing the following contributions:

RQ1 How can solution space artifacts be formally described in a meta-model?

> **Contribution**: A meta-model of the solution space is derived to serve as a theoretical foundation for further analysis methods of the solution space.

Benefits: In combination with a suitable meta-model for the problem space, combined and probably more efficient analysis methods for the whole PL can be developed.

RQ2 How can the realisability of a valid configuration be decided?

Contribution: A method to decide the realisability of a configuration is proposed. Based on that, two methods to decide about the consistency of problem and solution space are derived.

Benefits: The knowledge about configurations both valid (problem space) and realisable (solution space) reduces the amount of products which have to be analysed further in the development process. This allows for a resource-efficient PL testing.

RQ3 How can run-time variability be described?

Contribution: The meta-model from RQ1 is extended to express run-time dependencies of the solution space. Two demonstrations about their exemplary usage are presented.

Benefits: Analysis methods for existing products can contribute to more effective and efficient development and update routines for manufacturers.

3. Basics

This section covers the basic terms and concepts necessary for the contributions of this thesis. In Section 3.1, Product Lines (PLs) are introduced as they are the core concepts of this thesis. Further, related concepts which are based on PLs, such as configuration sampling and software product line engineering, are presented.

In Section 3.2, Constraint Satisfaction Problems (CSPs) are introduced to cover the basics of propositional logic problem statements. We use them to express and solve problems of instantiated meta-models mathematically, for example the validation of configurations or the assignment between software and hardware components.

3.1 Software Product Lines

The concepts of PLs were proposed by Northrop and Clemens as *Software Product Lines* cite [CN02]. It enables the development and production of product families which share a set of common artefacts and can be customized with variable artefacts. This enables the reuse of artefacts and the reduction of development costs. The following terms are taken from [PBvdL05].

The term *commonality* describes the set of artefacts which all products of the PL have in common. In context of software products, this can be a common implementation platform or core functionalities.

Commonalities can be extended and customized with the concept of *variability*. Variable artefacts are called *Variants*. Products of a PL differ in these variants. For software products, variants can be exchangeable software modules of different functionality.

The point where a commonality is extended by a variant is called *Variation Point*. The variation point describes the interface between commonality and variability.

Variability is further distinguished into variability in *space* and variability in *time* [PBvdL05]. Variability in space describes different variants addressing a single variation point. Variability in time describes different (development) versions of a variant. Besides variability in space, in this thesis we relate variability in time strongly to run-time variability and reduce the expression of modifications (to features or to the PL) to new feature or Feature Model (FM) versions.

Chapter 2 introduced an example PL from the automotive domain as our running example. The variability of this PL concerns the infotainment system where the variants `radio` and `navigation-system` are possible.

3.1.1 Feature Models & Configurations

To manage commonality and variability, a FM is used. It is a tree-like structure to express hierarchies and dependencies of a PL and all its possible products. The nodes (inner nodes and leafs) of a FM are called *Features*. A feature is a selectable, "[...] characteristic or end-user-visible behavior [...]" [ABKS13] of a (software) PL.

The FM is a recursive definition of parent- and child-features and can express five different types of hierarchies between them [ABKS13]. Figure 3.1 shows a generic FM with all five hierarchy types as listed in the following:

1. **Mandatory Feature**: A mandatory child-feature is selected, if its parent is selected. Mandatory features are marked with a filled circle above the feature name.
2. **Optional Feature**: An optional child-feature is selectable, if its parent is selected. Optional features are marked with an empty circle above the feature name.
3. **Or-Group**: At least one feature from the group of child-features of a parent has to be selected. Or-groups are marked with a filled triangle below the parent feature name of the group.

4. **Alternative-Group**: Exactly one feature from the group of child-features of a parent has to be selected. Alternative-groups are marked with an empty triangle below the parent feature of the group.

5. **Abstract Feature**: An abstract feature is a structural element of a FM without actual artefacts. Abstract features are marked with the stereotype-annotation <<`abstract`>>.

To express dependencies between features, for example between siblings or sub-trees, *Cross-tree Constraints* are used. Cross-tree constraints are arbitrary logical relations over features, for example for mutual exclusion of features (*excludes*-relation) or for requirements (*requires*-relation).

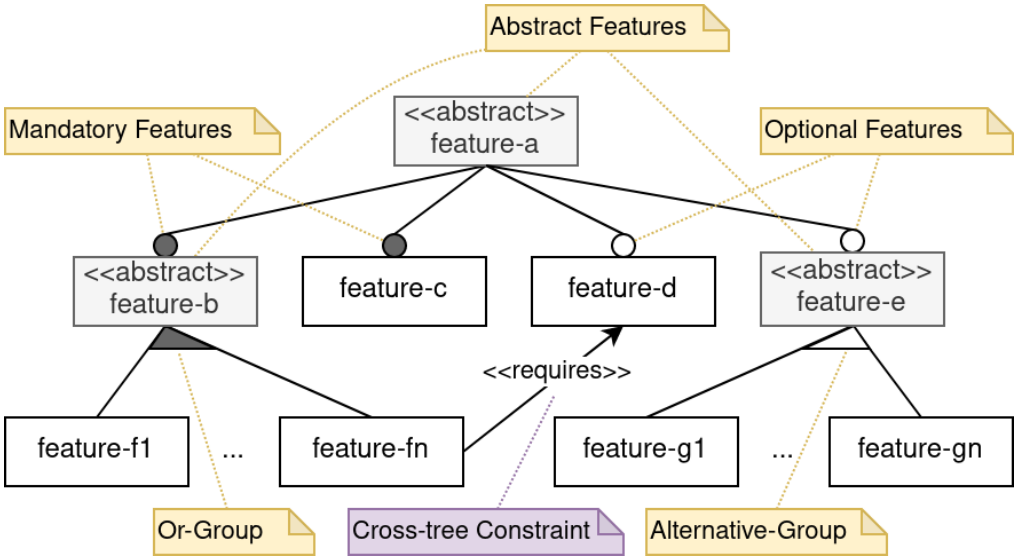

Figure 3.1: Five hierarchy types (yellow) and cross-tree constraint (purple) in a FM [ABKS13].

Figure 2.1 shows the FM for our running example. The abstract feature `info-tainment` is the parent of the alternative group containing the features `radio` and `navigation-system`. Therefore, exactly one of both options have to be selected. The feature `navigation-system` has an optional child-feature `real-time-traffic-system`, which can only be selected if its parent, the feature `navigation-system`, is selected. The running example has no cross-tree constraints.

A FM can also be expressed as a propositional logical formula which represents the structure of the tree [ABKS13]. Table 3.1 shows the translation of the tree-structures to logical terms. Each feature is a boolean variable which is **True** iff the feature is selected. A mandatory feature is selected equivalently to its parent ($\Leftrightarrow$-relation). The selection of an optional feature implies the selection of its parent ($\Rightarrow$-relation). If the parent of an or-group is selected, at least one of feature of the or group has to be selected. An alternative-group behaves similar to an or-group except it is assured that not more than one feature is selected.

Tree-Structure	Logical Term t
(root) feature r	t_r
mandatory feature m of parent p	$t_m \Leftrightarrow t_p$
optional feature o of parent p	$t_o \Rightarrow t_p$
or-group of features $\mathcal{F}_{\text{or}}$ of parent p	$\left(\bigvee_{f_i \in \mathcal{F}_{\text{or}}} t_{f_i} \right) \Leftrightarrow t_p$
alternative-group of features $\mathcal{F}_{\text{alt}}$ of parent p	$\left(\left(\bigvee_{f_i \in \mathcal{F}_{\text{alt}}} t_{f_i} \right) \Leftrightarrow t_p \right) \wedge \left(\bigwedge_{\substack{f_i, f_j \in \mathcal{F}_{\text{alt}} \\ f_i \neq f_j}} \neg(t_{f_i} \wedge t_{f_j}) \right)$

Table 3.1: Propositional logic representation of a FM. [ABKS13]

As a FM is a rooted tree, a single propositional logical formula can be derived. Cross-tree constraints are taken directly and conjuncted with the formula of the tree-structure. We use this logical representation of FMs in our contributions to apply formal analysis methods and give a concise definition in Chapter 4.

An actual selection of features in a FM is called *Configuration* [ABKS13]. A configuration can be an arbitrary subset of all features. A *valid* configuration is a configuration which fulfills all (hierarchical and cross-tree) constraints of the FM. To decide if a configuration is valid, the propositional logical representation of the FM is instantiated with the selection and deselection of the according features. Then, the decision if all constraints are fulfilled is equivalent to the validity of the configuration.

In the running example, the subset consisting of the features `radio` and `navigation-system` is a configuration, but not a valid configuration, because both features belong to an alternative-group where exactly one feature is allowed to be selected.

3.1.2 Configuration Sampling

The number of configurations in a PL can grow combinatorically (exponential upper boundary) to the number of features [TAK+14]. For example an or-group with n features and no further constraints generates $2^n - 1$ possible combinations. This is often referred to as *Combinatorial Explosion*. For large PLs, this can lead to an infeasible number of derivable products to test for the manufacturers, for which reason automated analysis methods of PLs were developed [BSRC10].

To overcome the challenges of testing an exponentially growing number of possible products, only a representative subset (a *Sample*) can be regarded. This concept is called *Configuration Sampling* and builds an own research field [VAHT+18]. This research examines topics such as which configurations exactly are in a sample and how the quality of samples can be derived.

One strategy after which configurations can be sampled is *Feature Interaction Coverage*. Feature interactions concern the combinations in which features can occur together with other features. This is of interest, as the combination of features can lead to unexpected behaviour with these features in the final product [ABKS13]. Feature interactions are classified along the combination sizes: 1-wise feature interactions regard only single features, pair-wise regard the combinations of two features,

etc. The coverage criteria guarantees that the sample contains, for example, all possible, valid pair-wise feature interactions.

Figure 3.2 shows the testing process for a sampled set of configurations. After the sampling, the manufacturer of the PL derives the products and tests them. Finally, the testing results of the samples are aggregated and implications on the complete product line can be stated.

In Chapter 4 we describe, how the contributions of this thesis (e.g. methods to decide about the consistency of the problem and solution space of a PL) can not only be applied to a full set of all valid configurations, but also on samples.

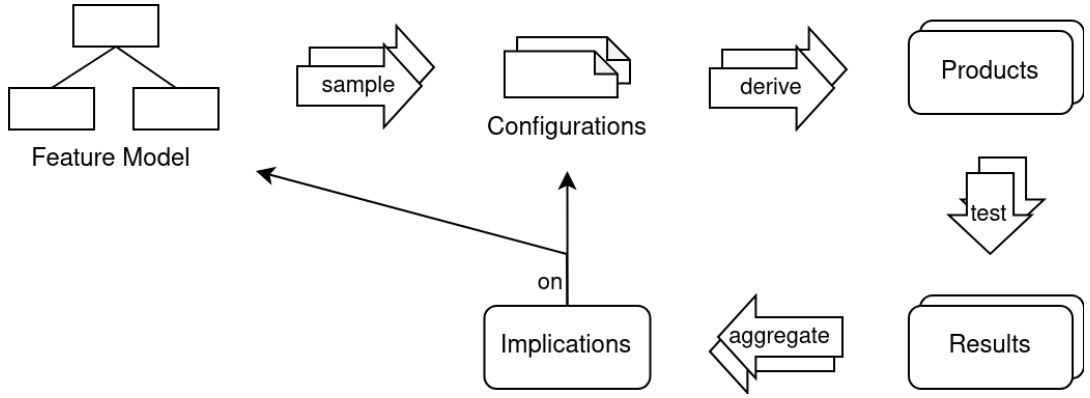

Figure 3.2: Testing process of a PL for a sampled set of configurations [ABKS13].

3.1.3 Software Product Line Engineering

The *Software Product Line Engineering* addresses the development architecture and processes of PLs [ABKS13]. It serves as a foundation to develop reusable software artefacts based on commonalities (software platform) and variabilities (feature artefacts) and defines roles, actions and artefacts.

Figure 3.3 shows the software product line engineering process defined by Apel et al. [ABKS13]. It is divided vertically into problem space and solution space. The *Problem Space* covers the requirements and the *Domain Analysis*, which concerns the management of features, design of the FM and generation of configurations. The *Solution Space* covers the realisation (implementation) and derivation of configurations as product variants.

We use the distinction between problem and solution as a central concept of the thesis. In the problem space, the PL is designed and its variability concepts are defined. To derive actual products, we address the solution space. With the contributions to the research questions, we propose a meta-model and an analysis method for the solution space.

The software product line engineering process is divided horizontally into domain and application engineering. The *Domain Engineering* addresses the whole PL in management, analysis and implementation terms with focus on developing reusable artefacts. The *Application Engineering* covers the requirement analysis and product derivation for customers and market segments with focus on using the reusable artefacts.

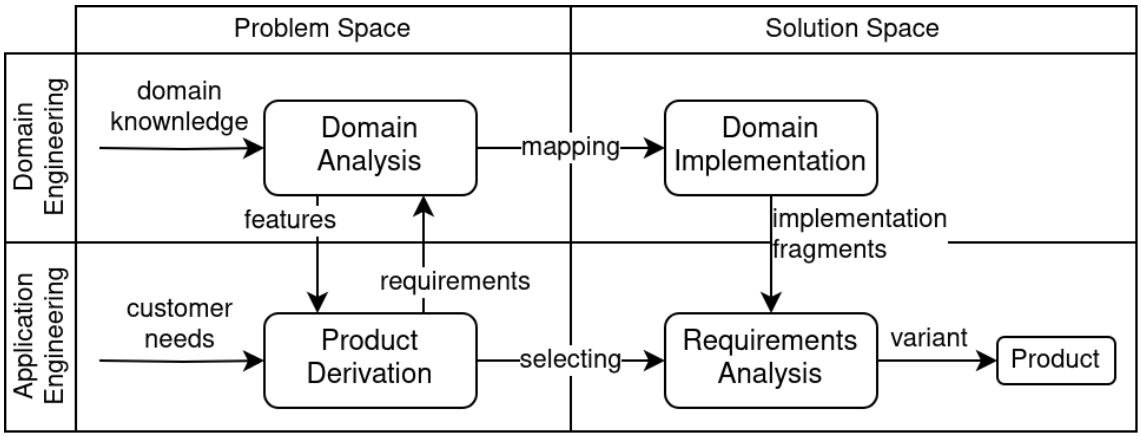

Figure 3.3: Software Product Line Engineering process adopted from [ABKS13].

3.1.4 Unified Conceptual Model

Ananieva et al. [AGK+22] proposed a meta-model of the problem space of a PL. It is called the Unified Conceptual Model (UCM) and shown in Figure 3.4. In the problem space, central concepts of variability in space (features and constraints) are unified with concepts of variability in time, such as feature and system revisions. In the problem space, products can be derived from configurations and solution space implementation "fragments".

Wittler et al. [WKR22] extended the UCM from Ananieva et al. [AGK+22] with a more concrete meta-model of the solution space. Figure 3.5 shows the complete version of Wittler et al. [WKR22]. A Product is derived from a configuration and solution space components. Components can require or provide generic services and are either software or hardware components. Further, software components demand resources and hardware components grant resources. A resource has a quantification and two properties: The property `isExclusive` determines if the resource is dedicated to a single software component or shared among multiple software components. The resource type defines meta-information, such as name and unit of measurement, and a comparison operator.

We use the UCM by Wittler et al. [WKR22] as a foundation to describe the solution space. Our contributions adapt the UCM and propose suitable representations to develop a method for deciding about the realisability of a configuration.

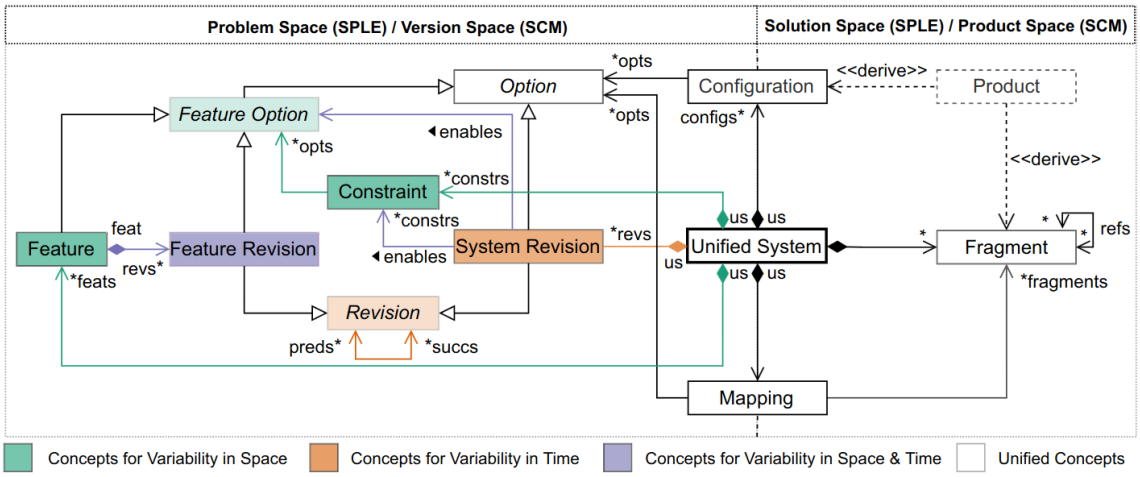

Figure 3.4: UCM from Ananieva et al. [AGK+22]

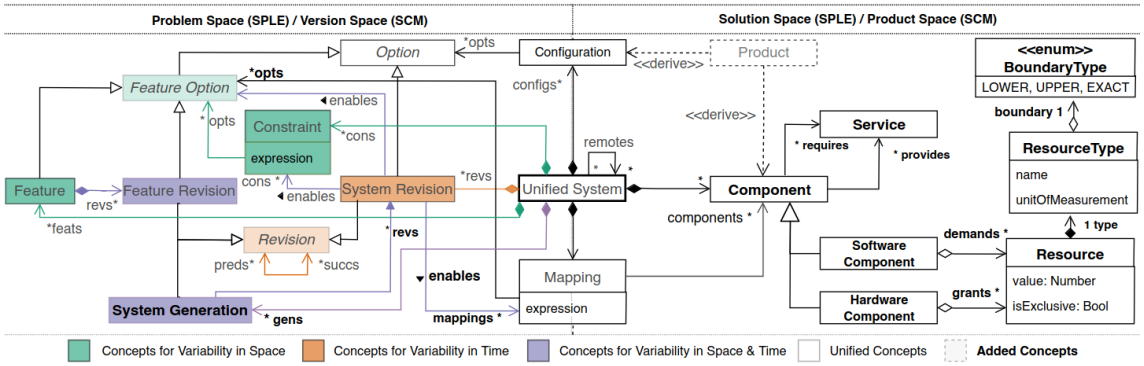

Figure 3.5: UCM from Wittler et al. [WKR22] with an extended meta-model of the solution space.

3.2 Constraint Satisfaction Problems

In propositional logic, a CSP is an approach to express a logical problem where a number of variables have to be assigned to allowed values in order to satisfy all stated constraints. The generality allows for a wide range of applications. We use specialised CSPs in our contributions to generate configurations which are both valid and realisable.

Definition

The definition is adopted from Russell et al. [RND10]. A CSP consists of a set of variables $\mathcal{X} := \{x_1, \ldots, x_n\}$ and a set of constraints $\mathcal{T}$ (in this thesis: $\mathcal{T}$ for "Terms"). Each variable has its own domain ($\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$) for the assigned value. The solution to a *satisfiable* CSP is an assignment from each variable to a value in its domain, called a *Satisfiability Model*. If no assignment is found that satisfy all constraints, the CSP is *unsatisfiable*.

In this thesis, we refer to the assignment of variables with the term *evaluation* and declare the general evaluation-function for a variable $x_i \in \mathcal{X}$ as $\text{eval}(x_i) \in \mathcal{D}_i$.

Each constraint t in the set of constraints $\mathcal{T}$ consists of two elements: A set of variables which occur in the constraint and a relation (e.g. a propositional logical formula) that defines, which values the variables can be assigned to in order to satisfy the constraint. In this thesis, we typically omit the first element of a constraint (the set of participating variables) and directly specify a propositional logical formula for reasons of simplicity.

Application Example

An example of a CSP is the generation of a valid configuration from a FM. Features are defined as variables in the boolean domain $\mathbb{B}$, indicating if a feature f is selected in the configuration ($\text{eval}(f) = \textbf{True}$) or not selected ($\text{eval}(f) = \textbf{False}$). The set of constraints correspond to the propositional logical representation of the FM introduced in Subsection 3.1.1.

If the formalised CSP is satisfiable, the solution is one (of possibly many) valid configurations. The satisfiability model expresses for each feature if it is selected in the configuration or not selected. If the formalised CSP is unsatisfiable, no valid configuration exists and the FM is considered as inconsistent [ABKS13].

Resolvement & Generalisation of Constraint Satisfaction Problems

CSPs are typically solved by Commercial Off-The-Shelf (COTS) tools. The selection of such a tool depends on the domains of the CSPs variables and the complexity of its constraints. A CSP with boolean variables (domain $\mathbb{B}$) and boolean propositional logic (operators $\wedge$, $\vee$ and $\neg$) can be solved by Boolean Satisfiability Problem (SAT)-solvers cite [GZ17]. For generalised problems, e.g. with real values (domain $\mathbb{R} \supset \mathbb{B}$) and more complex constraints containing arithmetics, other COTS tools have to be chosen, for example pseudo-boolean solvers [BH02] or Satisfiability Modulo Theories (SMT)-solvers [DMB11]. As CSPs over FMs are computationally hard to solve (NP complexity, [ABKS13]), there exist approaches to reduce the complexity to polynomial algorithms [Mil72]. Tailoring CSPs to special use cases can also simplify their computational complexity by loosing generality.

4. Design

In this chapter, the solution approaches of the thesis are introduced. Addressing RQ1, Section 4.1 proposes a meta-model to describe the solution space. This meta-model serves as a foundation for further contributions of this thesis.

A formal method to decide the realisability of a valid configuration (RQ2) is proposed in Section 4.2. Then, the underlying scope is generalised to the decision about the consistency of the problem and solution space in Section 4.3. Two different methods are presented to approach the problem: A method based on configuration sampling (Subsection 4.3.1) and a method that combines formal analysis methods of the problem and solution space (Subsection 4.3.2).

In Section 4.4, a description of run-time variability is proposed and suggestions for the handling of run-time variability are made. A summary of the contributions is given in Section 4.5.

Preliminaries

The remainder of this document uses mathematical denotions to express statements more precisely. For concepts introduced in Chapter 3, the according denotions are defined in this paragraph. Later, presented concepts come directly with their denotion. In context of this document, we use the following symbols:

$\mathcal{F}$ **The set of all features.**

f **A feature.**

A feature is part of the set of features $\mathcal{F}$. Usually, a feature is indexed by the control variable l. Example: f_1

$\mathcal{CS}$ **The set of all valid configurations.**

The set of all valid configurations is a subset of all possible combinations of features ($\mathcal{CS} \subseteq \mathcal{P}(\mathcal{F})$).

$\mathcal{C}$ **A configuration.**

A configuration is part of the set of configurations $\mathcal{CS}$. Specific configurations, like these of the running example, are indexed lexicographically. Example: $\mathcal{C}_{\mathrm{A}}$.

$\mathcal{RS}$ **The set of all valid and realisable configurations.**

The set of all valid and realisable configurations is a subset of all valid configurations ($\mathcal{RS} \subseteq \mathcal{CS}$).

$\mathcal{SW}$ **The set of all software components.**

sw **A software component.**

A software component is part of the set of software components $\mathcal{SW}$. Usually, a software component is indexed by the control variable i. Example: sw_0.

$\mathcal{HW}$ **The set of all hardware components.**

hw **A hardware component.**

A hardware component is part of the set of hardware components $\mathcal{HW}$. Usually, a software component is indexed by the control variable j. Example: hw_0.

$\mathcal{T}$ **A set of propositional logical terms.**

t **A propositional logical term.**

A propositional logical term can be a single logical variable of a defined type or a logical-connected proposition of terms. Example: $t_0 \wedge t_1$.

$\mathbb{B}$ **The boolean domain.** ($\mathbb{B} := \{\mathbf{False}, \mathbf{True}\}$)

In context of this document, we define the following mathematical functions and operators:

sgn **Signum Functions**

The family of signum functions consists of two different characteristics:

- From *boolean to numeric* domain

$$\mathrm{sgn}_{\mathbb{B}\to\mathbb{Z}} : \mathbb{B} \to \{0, 1\} \subset \mathbb{Z}$$
$$\mathrm{sgn}_{\mathbb{B}\to\mathbb{Z}}(x) = \begin{cases} 1 & \text{if } x = \mathbf{True} \\ 0 & \text{else} \end{cases}$$

- From *numeric to boolean* domain (zero included)

$$\mathrm{sgn}_{\mathbb{R}_0\to\mathbb{B}} : \mathbb{R} \to \mathbb{B}$$
$$\mathrm{sgn}_{\mathbb{R}_0\to\mathbb{B}}(x) = \begin{cases} \mathbf{True} & \text{if } x \geq 0 \\ \mathbf{False} & \text{else} \end{cases}$$

sel **Select Function**

The select function determines if a feature is selected in context of a configuration.

$$\text{sel} : \mathcal{F} \to \mathbb{B}$$
$$\text{sel}(f) = \begin{cases} \textbf{True} & \text{if the feature } f \text{ is selected in the configuration} \\ \textbf{False} & \text{if the feature } f \text{ is } \textit{not} \text{ selected in the configuration} \end{cases}$$

eval **Evaluation Function**

The evaluation function computes the truth-value of a term.

$$\text{eval} : \mathcal{T} \to \mathbb{B}$$

$\equiv_{\mathcal{C}}$ **Equality of Two Configurations**

This function determines, if two configurations are equal. It is used with infix-notation.

$$\equiv_{\mathcal{C}} : \mathcal{CS}^2 \to \mathbb{B}$$
$$\mathcal{C}_\text{A} \equiv_{\mathcal{C}} \mathcal{C}_\text{B} \Leftrightarrow \begin{cases} \textbf{True} & \text{if both configurations have the same features selected} \\ \textbf{False} & \text{else} \end{cases}$$

$\equiv_{\mathcal{CS}}$ **Equality of Two Sets of Configurations**

This function determines, if two sets of configurations are equal. They are equal, if they have the same size and *exactly equal* elements. The function is used with infix-notation.

$$\equiv_{\mathcal{CS}} : \mathcal{P}(\mathcal{F})^2 \to \mathbb{B}$$
$$\begin{aligned} &\mathcal{CS}_A \equiv_{\mathcal{CS}} \mathcal{CS}_B \\ \Leftrightarrow\ &|\mathcal{CS}_A| = |\mathcal{CS}_B| \\ &\wedge\ \forall \mathcal{C}_i^\text{A} \in \mathcal{CS}_A : \exists \text{M}_i : \left(\text{M}_i := \{\mathcal{C}_j^\text{B} \in \mathcal{CS}_B \,|\, \mathcal{C}_j^\text{B} \equiv_{\mathcal{C}} \mathcal{C}_i^\text{A}\} \wedge |\text{M}_i| = 1\right) \end{aligned}$$

4.1 Description of the Solution Space

The Unified Conceptual Model (UCM) from [WKR22] is used as a formal description of Product Lines (PLs). This meta-model fits as a starting point of the thesis for several reasons: First, it contains the problem space and the solution space of the PL. Second, it assigns concepts clearly to either the problem space, the solution space or the link between them. Third, the solution space of the UCM provides a resource- and service-based view of implementation artefacts. This allows a fine-grained management of solution space artefacts and refers to recently approached system architectures of Cyber-Physical Systems (CPSs) [SSG+22, SDS04]. Last, as the thesis also deals with run-time variability (RQ3), the concept of feature revisions and system revisions enable the description of versions, thus development, of features and Feature Models (FMs).

In context of this thesis, the solution space of the UCM is further modified in Subsection 4.1.1. An approach to describe actual resource demands in the UCM with an attributed FM is given in Subsection 4.1.2. Details of the linkage of problem and solution space are proposed in Subsection 4.1.3.

4.1.1 Modification of the Unified Conceptual Model

The UCM is modified to suit the needs of further work of this thesis. In detail, the following adaptions are made:

1. The UCM, as a Unified Modeling Language (UML) class diagram, formerly describes a property `isExclusive` of the class `Resource`.

   Modification: The property `isExclusive` is moved from the class `Resource` to the class `ResourceType`.

   Intention: In context of this thesis, all properties of resources should be defined via the type of the resource. It could be argued that in some cases, a concrete instance of a `Resource` should behave different from other instances of the same type. These cases are neglected as they introduce more complexity into the problem and could potentially be omitted by defining another resource type with the desired properties.

2. **Modification**: The class `ResourceType` is extended by the property `isAdditive` with type `Bool`.

   Intention: Not all physical resources behave equally in case of multiple occurrence. For example, if a Computation Unit (CU) has a response time of 1 ms, this value can still be used with multiple, equal CUs or multiple threads running on the same CU. Considering Random Access Memory (RAM) on the other hand, the capacity values add up. The introduced property `isAdditive` supports this consideration and enables the distinction between ”additive” and ”non-additive” resources.

3. The class `Resource` is associated with the class `ResourceType` by a composition.

   Modification: The association will be an aggregation.

Intention: After the modifications 1 and 2, the class `Resource` serves as a quantification object of an instance of the class `ResourceType` to an instance of the class `Component`. It is intended that the type of a resource can be specified regardless of its quantification.

4. **Modification**: The class `Hardware Component` has an aggregation to the class `Resource` renamed to `provides`.

In this document, we denote a resource type with r. Usually, it is identified and indexed by the control variable k. The set of all resource types is denoted by $\mathcal{R}$. The values of the properties of a resource type (`isAdditive`, `isExclusive`, `boundary`) are denoted with the v-function (v for *value*), which is declared in Equation 4.1.

$$\begin{aligned} \mathrm{v}_{\mathrm{add}} &: \mathcal{R} \to \mathbb{B} \\ \mathrm{v}_{\mathrm{exc}} &: \mathcal{R} \to \mathbb{B} \\ \mathrm{v}_{\mathrm{bnd}} &: \mathcal{R} \to \{\texttt{LOWER}, \texttt{UPPER}, \texttt{EXACT}\} \end{aligned} \tag{4.1}$$

Figure 4.1 shows the resulting version of the UCM. Table 4.1 lists the specified resource types for the running example (Chapter 2) as instances of the class `ResourceType`. First, a resource type for RAM with additive, non-exclusive behaviour and lower boundary type is defined (r_0). For $k = 1$, a resource type for the response time of a component is defined with non-additive, non-exclusive behaviour and upper boundary type. Last, a resource type for the number of screens attached to the component is specified with additive, non-exclusive behaviour and exact boundary type.

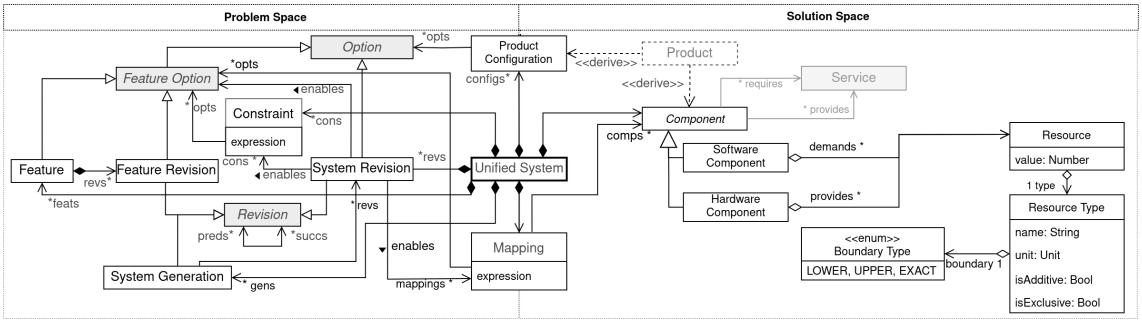

Figure 4.1: The UCM from Wittler et al. [WKR22] after initial modifications. The part of services is addressed later.

k	name	unit	isAdditive	isExclusive	boundary
0	RAM	GB	true	false	LOWER
1	Response Time	ms	false	false	UPPER
2	no. of Screens		true	false	EXACT

Table 4.1: Three instances of the class `ResourceType`, defined for the running example introduced in Chapter 2.

4.1.2 Description of Resource Demands in Attributed Feature Model

The modified UCM gives the ability to quantify the resource demands of software components and the resource provisionings of hardware components. We will now propose, how demands can directly be written to features. With attributed FMs, features can be enriched with (meta-) information [BTRC05].

Although information about resource quantification is applied in the solution space, it is decided to shift the information into the problem space for the following reasons: First, the actual resource demands of a configuration depend on the selected feature. Therefore, managing these demands can be seen as a task of domain and application engineering. Both topics are part of the problem space as proposed by [ABKS13]. Furthermore, resource demands of features can change over time as the features itself are going to be updated. With new feature revisions, the according resource demands are also versioned [AGK+22].

Each feature $f_l \in \mathcal{F}$, that specifies any resource demands, has to be attributed with the following information per resource demand:

1. It must be given, which software component $sw_i \in \mathcal{SW}$ is addressed. Software components serve as an abstraction layer and allow an independent encapsulation of resource demands. Hence, a feature can map to multiple software components.

2. It must be given, which resource type $r_k \in \mathcal{R}$ is addressed. This information is treated in context of the specified software component sw_i. Hence, a feature can have resource demands for multiple resource types.

3. The quantification of the resource demand must be given. This information is treated in context of the specified software component sw_i and resource type r_k. For the time of the plain concept of the solution space description, the allowed values shall not be restricted further and are assumed as a numeric type (generally the domain of real numbers $\mathbb{R}$).

In this document, such a resource demand is denoted with $rd^i_{k_l}$. For the running example, three resource types were defined in Table 4.1. The actual resource demands of the features are illustrated in Figure 4.2 with an attributed FM. An edge from a feature f_l to a resource type r_k of a software component sw_i describes a resource demand $rd^i_{k_l}$. The annotated edge weight quantifies the resource demand. It is possible that a feature has resource demands to multiple resource types and (especially feature `real-time-traffic-system`) to multiple software components.

4.1.3 Mapping Between Features and Software Components

The attributed FM describes resource demands $rd^i_{k_l}$ of features $f_l \in \mathcal{F}$ in context of a software component $sw_i \in \mathcal{SW}$ and a resource type $r_k \in \mathcal{R}$. Hence, from the view point of a software component sw_i, there exists a set of resource demands from features $(f_1, f_2, \ldots, f_{|F|})$ in context of a resource type r_k. We denote such a set with $\mathcal{RD}^i_k := \{rd^i_{k_1}, rd^i_{k_2}, \ldots, rd^i_{k_{|F|}}\}$.

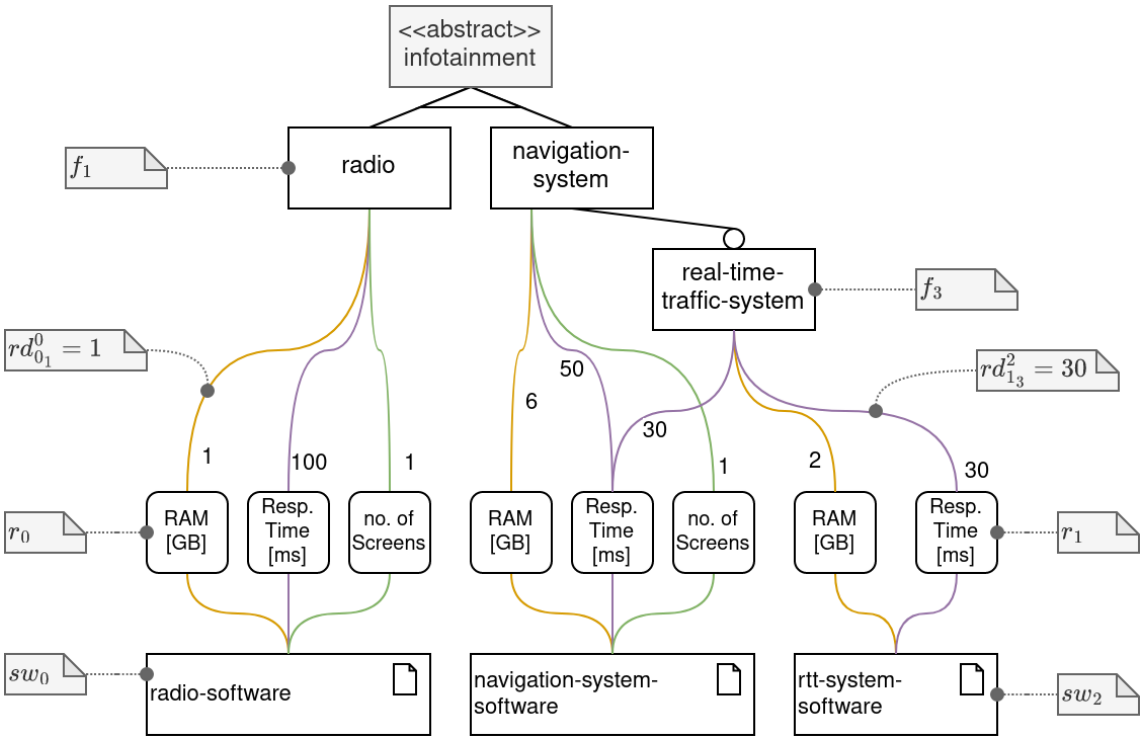

Figure 4.2: Attributed FM of the running example. Edges from features to resource types of software components illustrate the resource demands.

It is bounded by $0 \leq |\mathcal{RD}^i_k| \leq |\mathcal{F}|$: In case none of the features specify any resource demand of resource type r_k to the software component sw_i, the lower boundary is given and vice versa if all features specify such a resource demand.

According to the UCM, a software component sw_i quantifies a resource demand of a resource type r_k with a numeric type. This resource demand is denoted by rd^i_k. To map the set $\mathcal{RD}^i_k$ to rd^i_k, the *reduce*-function is defined in Equation 4.4. The function applies to resource demands of *selected* features only, denoted by $\overline{\mathcal{RD}^i_k} = \{rd^i_{k_l} \,|\, \text{sel}(f_l) = \textbf{True}\}$. This implies that a concrete configuration $\mathcal{C} \subseteq \mathcal{F}$ has to be known in advance. The mapping between features and software components therefore is part of the *process* to derive solution space artefacts and decide the realisability of configurations (further described in Section 4.2).

In context of a resource type r_k having the property `isAdditive` set to **True** ($\text{v}_{\text{add}}(r_k) = \textbf{True}$), the sum of all specified resource demands $rd^i_{k_l}$ is taken. If not, a comparison based on the boundary type of the resource type (`LOWER`, `UPPER`, `EXACT`) is done. For the boundary type `EXACT`, it is assumed that all resource demands specify the same value. Then, the *fst*-function is used, which returns the first element of a set and is defined in Equation 4.2. Otherwise, it can not be decided which value has to be taken. For an empty set, nil is returned. In this document, nil is the specification of *nothing*. To determine if a resource demand of a software component for a resource type is given (any value) or not given (*nothing*), we define the nil-function and its negation ($\overline{\text{nil}}$) in Equation 4.3.

$$\begin{aligned} &\text{fst} : \mathcal{P}(\mathbb{R}) \to \mathbb{R} \\ &\text{fst}(X) = x_0 \quad \left(x_0 \in \langle x_i \mid x_i \in X, i \in \{0, \ldots, |X| - 1\}\rangle\right) \end{aligned} \tag{4.2}$$

$$\begin{aligned} &\mathrm{nil}, \overline{\mathrm{nil}} : \mathcal{RD} \to \mathbb{B} \\ &\mathrm{nil}(rd_k^i) = \begin{cases} \textbf{True} & \text{if } rd_k^i \neq \perp \\ \textbf{False} & \text{else} \end{cases} \\ &\overline{\mathrm{nil}}(\cdot) = \neg \mathrm{nil}(\cdot) \end{aligned} \tag{4.3}$$

$$\mathrm{red}(\overline{\mathcal{RD}_k^i}) = \begin{cases} \sum_{f_l \in \mathcal{C}} rd_{k_l}^i & \text{if } \mathrm{v}_{\mathrm{add}}(r_k) = \textbf{True} \\ \max(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{LOWER} \\ \min(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{UPPER} \\ \mathrm{fst}(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{EXACT} \\ \perp & \text{if } |\overline{\mathcal{RD}_k^i}| = 0 \text{ and else} \end{cases} \tag{4.4}$$

Figure 4.2 introduces the attributed FM of the running example. Regarding the mapping process, the resource type for the response time (r_1) of the software component `navigation-system-software` (sw_1) is detailed in Figure 4.3. Both the features `navigation-system` (f_2) and `rtt-system-software` (f_3) are selected (configuration $\mathcal{C}_\mathrm{C}$). As the resource type r_1 is non-additive and has the boundary type `UPPER`, the min function is used to reduce the demands $rd_{1_2}^1 = 50$ and $rd_{1_3}^1 = 30$ to $rd_1^1 = 30$. Additionally, it can be seen that the mapping is placed in the middle between problem and solution space, following the UCM. While the resource demands from the features (attributed FM) are held in the problem space, the mapped resource demand $rd_1^1 = 30$ belongs to the solution space artifact `navigation-system-software` (sw_1).

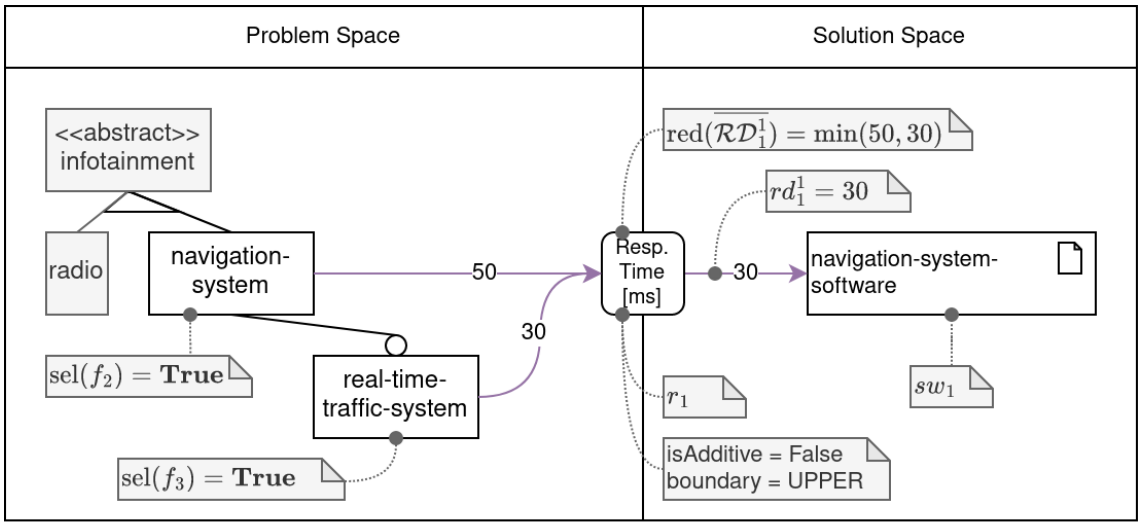

Figure 4.3: Mapping detail of configuration $\mathcal{C}_\mathrm{C}$ of the running example: The resource demands $rd_{1_2}^1 = 50$ (feature f_2) and $rd_{1_3}^1 = 30$ (feature f_3) from the problem space are mapped to $rd_1^1 = 30$ in the solution space.

4.2 Decide the Realisability of Configurations

The solution space of a PL is described by the UCM as defined in Figure 4.1. For a configuration, solution space artefacts can be derived and consist of software components, which demand resources, and hardware components, which provide resources. To decide the realisability of a configuration, it must be decided if all of the demanded resources are provided by the hardware components. In the literature, such a computational problem is often referred to as a *Resource Allocation Problem* or, more generalised, as an *Assignment Problem* [KI98, Law01].

Typically, an assignment problem is formalised as an optimisation problem. Therefore, an optimisation function is proposed which is going to be maximised or minimised. In the scope of this thesis, the optimisation can be omitted, as only the decision if *any* solution is possible, is of interest. Hence, no optimisation function is needed.

In this Section, first such an assignment problem is formalised for the given metamodel of the solution space in Subsection 4.2.1. Then the complete procedure to decide the realisability of a configuration is explained along the running example in Subsection 4.2.2. Implications of the results of solving the assignment problem are defined in Subsection 4.2.3.

4.2.1 Formalising the Solution Space Assignments as CSP

The assignment of resource demands to resource provisionings can be formalised and solved as a Constraint Satisfaction Problem (CSP). Constants and logical variables form logical constraints. These constraints are joined together by logical conjunctions ($\wedge$-Operator). In the following, the definitions of inputs, variables, constraints and assembly of the CSP are presented.

4.2.1.1 Input

The input of the CSP is mainly driven by three sets:

1. **Resource Types** $\mathcal{R}$: A set of instances of the `ResourceType` class (see Figure 4.1). Each resource type r_k ($\forall k \in \{0, \ldots, |\mathcal{R}| - 1\}$, $r_k \in \mathcal{R}$) has its three properties (`isAdditive`, `isExclusive`, `boundary`) defined as constants. Therefore, this input is of size $|R| \times 3$.

2. **Resource Demands** $\mathcal{RD}$: For each software component sw_i ($\forall i \in \{0, \ldots, |\mathcal{SW}| - 1\}$, $sw_i \in \mathcal{SW}$) the resource demands rd_k^i for all resource types r_k are specified as constants. If a resource demand for a specified software component $sw_{i'}$ and resource type $r_{k'}$ is not set (e.g., because $|\overline{\mathcal{RD}}_{k'}^{i'}| = 0$), it is set to nil ($rd_{k'}^{i'} = \perp$). This input is of size $|\mathcal{SW}| \times |\mathcal{R}|$.

3. **Resource Provisionings** $\mathcal{RP}$: For each hardware component hw_j ($\forall j \in \{0, \ldots, |\mathcal{HW}| - 1\}$, $hw_j \in \mathcal{HW}$) the resource provisionings rp_k^j for all resource types r_k are specified as constants. If a resource provisioning for a specified hardware component $hw_{j'}$ and resource type $r_{k'}$ is not set (e.g., because the hardware component does not provide this type of resource), it is set to nil ($rp_{k'}^{j'} = \perp$). This input is of size $|\mathcal{HW}| \times |\mathcal{R}|$.

4.2.1.2 Variables & Constraints

The goal of the CSP is to assign each resource demand to a resource provisioning. As this assignment is done per resource type, a three-dimensional set of resource-wise assignment variables is defined in Equation 4.5.

$$\begin{aligned}&\forall i \in \{0,\dots,|\mathcal{SW}|-1\} : \forall j \in \{0,\dots,|\mathcal{HW}|-1\} : \forall k \in \{0,\dots,|\mathcal{R}|-1\} :\\ &\text{eval}(rz_k^{i,k}) = \begin{cases}\textbf{True} & \text{if } rd_k^i \text{ of } sw_i \text{ is assigned to } rp_k^j \text{ of } hw_j \text{ in context of } r_k\\ \textbf{False} & \text{else}\end{cases}\end{aligned} \tag{4.5}$$

For further global constraints, it is necessary to not only define resource-wise assignment variables, but also component-wise assignment variables. These enforce the assignment of a software component to a single hardware component over all resource types and are defined in Equation 4.6.

The reason for this design decision is to strengthen the concept of components as an abstraction layer. As it encapsulates a set of resource demands (in case of a software component), it can be argued that all of these resource demands should be satisfied by the resource provisionings of exactly one hardware component. Furthermore, it enables a cyber-spatial control over the deployment of software components to hardware components: For example should the necessary amount of RAM be provided by the same hardware component than the also necessary Arithmetic Logic Unit (ALU) regarding latency. By enforcing component-wise assignments, this technical requirement can be designed.

For the software components to be actually assigned to exactly one hardware component each, Equation 4.7 defines the necessary constraint.

$$\begin{aligned}&\forall i \in \{0,\dots,|\mathcal{SW}|-1\} : \forall j \in \{0,\dots,|\mathcal{HW}|-1\} :\\ &cz^{i,k} \Leftrightarrow \bigwedge_{k=0}^{|\mathcal{R}|-1} rz_k^{i,j}\end{aligned} \tag{4.6}$$

$$\forall i \in \{0,\dots,|\mathcal{SW}|-1\} : \sum_{j=0}^{|\mathcal{HW}|-1} \text{sgn}_{\mathbb{B}\to\mathbb{Z}}(\text{eval}(cz^{i,j})) = 1 \tag{4.7}$$

Finally, constraints which define the satisfaction of a resource demand have to be introduced. These constraints depend on the hardware component hw_j, to which the provided resources belong, and the resource type r_k, for which the resource demand has been stated. For example, resource demands in context of an additive resource type behave differently to resource demands in context of a non-additive resource type.

Each resource type r_k holds information about its three properties (`isAddtitive`, `isExclusive`, `boundary`). Regarding these properties, there are twelve characteristics possible in total: Two binary properties (`isAddtitive`, `isExclusive`) and one ternary property (`boundary`). The according constraints are denoted by

$t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathrm{v}_{\mathrm{exc}}(r_k),\mathrm{v}_{\mathrm{bnd}}(r_k)}$ with the properties in this order. To identify an actual value of a property, it is sub-scripted with the short name of the according property, for example $t_{\mathbf{True}_{\mathrm{add}},\mathbf{True}_{\mathrm{exc}},\mathtt{LOWER}_{\mathrm{bnd}}}$.

Instead of defining constraints for each of these twelve characteristics, the following two implications can be made:

1. An exclusive resource type can be expressed with a conjunction of its non-exclusive equivalent and an upper bound of one, regarding the number of assigned software components which specify a resource demand. Equation 4.8 shows the formalisation for a fixed hardware component hw_j and a fixed resource type r_k.

$$\begin{aligned} & t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathbf{True}_{\mathrm{exc}},\mathrm{v}_{\mathrm{bnd}}(r_k)} \\ \Leftrightarrow\ & t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathbf{False}_{\mathrm{exc}},\mathrm{v}_{\mathrm{bnd}}(r_k)} \\ & \wedge \left(\sum_{i=0}^{|\mathcal{SW}|-1} \left(\mathrm{sgn}_{\mathbb{B}\to\mathbb{Z}}\left(\mathrm{eval}(rz_k^{i,j}) \wedge \overline{\mathrm{nil}}(rd_k^i) \right) \right) \leq 1 \right) \end{aligned} \tag{4.8}$$

2. The boundary types `LOWER` and `UPPER` are assumed to be non-strict ($\geq$- and $\leq$-operators). Then, the boundary type `EXACT` can be expressed with a conjunction of the boundary types `LOWER` and `UPPER`, as formalised in Equation 4.9.

$$\begin{aligned} & t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathrm{v}_{\mathrm{exc}}(r_k),\mathtt{EXACT}_{\mathrm{bnd}}} \\ \Leftrightarrow & t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathrm{v}_{\mathrm{exc}}(r_k),\mathtt{LOWER}_{\mathrm{bnd}}} \wedge t_{\mathrm{v}_{\mathrm{add}}(r_k),\mathrm{v}_{\mathrm{exc}}(r_k),\mathtt{UPPER}_{\mathrm{bnd}}} \end{aligned} \tag{4.9}$$

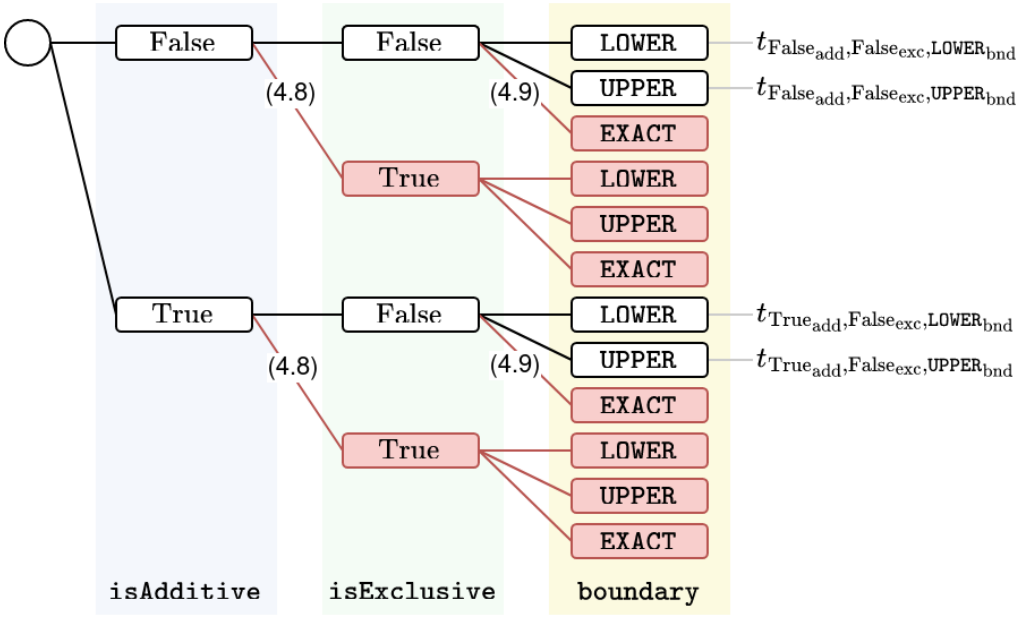

Figure 4.4: Initial twelve characteristics of resource type properties. Due to two logical implications (Equation 4.8, Equation 4.9), only four constraints have to be made. The red paths are pruned.

Figure 4.4 shows the initial twelve characteristics of resource type properties. The implications reduce the number of different characteristics to four: First, they are reduced to six as only non-exclusive resource types have to be regarded further. Second, the number is reduced to four, as only two of three possible boundary types have to be regarded further. These four constraints are defined in Equation 4.10 to Equation 4.13 for a fixed hardware component hw_j and a fixed resource type r_k.

For resource types which are non-additive and lower bounded ($t_{\mathbf{False}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{LOWER}_{\text{bnd}}}$; Equation 4.10), it is required that the provided resources are at least as big as demanded resources. This holds analogously, for additive, lower-bounded resource types ($t_{\mathbf{True}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{LOWER}_{\text{bnd}}}$; Equation 4.12), but in context of summed up values. For non-additive, upper-bounded resource types ($t_{\mathbf{False}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{UPPER}_{\text{bnd}}}$; Equation 4.11) it is required that the provided resources do not exceed the demanded resources. Analogously for additive, upper-bounded ($t_{\mathbf{True}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{UPPER}_{\text{bnd}}}$; Equation 4.13) resource types, but in context of summed up values. The nil-function and eval-function are used to regard only given resource demands of assigned components in the calculations.

$$t_{\mathbf{False}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{LOWER}_{\text{bnd}}} \Leftrightarrow \bigwedge_{i=0}^{|\mathcal{SW}|-1} \operatorname{sgn}_{\mathbb{R}_0\to\mathbb{B}}(rp_k^j - rd_k^i \cdot \operatorname{sgn}_{\mathbb{B}\to\mathbb{Z}}(\operatorname{eval}(rz_k^{i,j}))) \tag{4.10}$$

$$t_{\mathbf{False}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{UPPER}_{\text{bnd}}} \Leftrightarrow \bigwedge_{i=0}^{|\mathcal{SW}|-1} \operatorname{sgn}_{\mathbb{R}_0\to\mathbb{B}}(rd_k^i - rp_k^j \cdot \operatorname{sgn}_{\mathbb{B}\to\mathbb{Z}}(\operatorname{eval}(rz_k^{i,j}) \wedge \overline{\operatorname{nil}}(rd_k^i)) \tag{4.11}$$

$$t_{\mathbf{True}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{LOWER}_{\text{bnd}}} \Leftrightarrow \sum_{i=0}^{|\mathcal{SW}|-1} (rd_k^i \cdot \operatorname{sgn}_{\mathbb{B}\to\mathbb{Z}}(\operatorname{eval}(rz_k^{i,j}))) \leq rp_k^j \tag{4.12}$$

$$\begin{aligned} t_{\mathbf{True}_{\text{add}},\mathbf{False}_{\text{exc}},\texttt{UPPER}_{\text{bnd}}} \Leftrightarrow & \sum_{i=0}^{|\mathcal{SW}|-1} (rd_k^i \cdot \operatorname{sgn}_{\mathbb{B}\to\mathbb{Z}}(\operatorname{eval}(rz_k^{i,j}) \wedge \overline{\operatorname{nil}}(rd_k^i))) \\ & \geq rp_k^j \cdot \operatorname{sgn}_{\mathbb{B}\to\mathbb{Z}}(\bigvee_{i=0}^{|\mathcal{SW}|-1} (\operatorname{eval}(rz_k^{i,j}) \wedge \overline{\operatorname{nil}}(rd_k^i))) \end{aligned} \tag{4.13}$$

4.2.1.3 Assembling

To assemble the CSP, all constraints are conjoined together. The method is illustrated in Listing 4.1.

First, an empty set of constraints $\mathcal{T}$ has to be initialized (line 5). Then the boolean variables for the resource-wise assignments rz (line 7 to 10, Equation 4.5) and component-wise assignments cz (line 12 to 14) are declared. The connections between those two assignment variables, which were introduced in Equation 4.6, are added to the set of constraints $\mathcal{T}$ (line 16 to 18). The constraints to enforce the

assignment of each software component to exactly one hardware component (introduced in Equation 4.7) are added in line 20 to 21. Last, the constraints which define the resource demand satisfaction are added in line 23 to 25, referring to Equation 4.10, Equation 4.11, Equation 4.12 and Equation 4.13. The completed problem is conjuncted in line 27.

```
1  R resource types (input constants)
2  RP resource provisionings (input constants)
3
4  function csp(RD: resource demands (input constants)):
5      T ← ∅ set of constraints
6
7      ∀i ∈ {0, ..., |SW| − 1} :
8          ∀j ∈ {0, ..., |HW| − 1} :
9              ∀k ∈ {0, ..., |R| − 1} :
10                 declare boolean variable rz_k^{i,j}
11
12     ∀i ∈ {0, ..., |SW| − 1} :
13         ∀j ∈ {0, ..., |HW| − 1} :
14             declare boolean variable cz^{i,j}
15
16     ∀i ∈ {0, ..., |SW| − 1} :
17         ∀j ∈ {0, ..., |HW| − 1} :
18             T ∪ {cz^{i,k} ⇔ ⋀_{k=0}^{|R|−1} rz_k^{i,j}}
19
20     ∀i ∈ {0, ..., |SW| − 1} :
21         T ∪ {∑_{j=0}^{|HW|−1} sgn_{B→Z}(eval(cz^{i,j})) = 1}
22
23     ∀j ∈ {0, ..., |HW| − 1} :
24         ∀k ∈ {0, ..., |R| − 1} :
25             T ∪ {t_{v_add(r_k), v_exc(r_k), v_bnd(r_k)}}
26
27     return ⋀(T)
```

Listing 4.1: Assembly of the constraint set $\mathcal{T}$ and the complete CSP to decide the realisability of a configuration.

4.2.2 Procedure

This subsection describes the procedure to decide the realisability of a configuration. As a prerequisite, an annotated FM with a defined set of features $\mathcal{F}$ and resource types $\mathcal{R}$ must be known in advance. Figure 4.5 illustrates the following steps of the procedure:

1. A valid configuration is derived from the annotated FM. This configuration yields the resource demand sets $\mathcal{RD}_k^i$ for each software component $sw_i \in \mathcal{SW}$ and each resource type $r_k \in \mathcal{R}$ from the features $f_l \in \mathcal{C} \subseteq \mathcal{F}$.

2. With the reduce function, the mapping of each resource demand set $\mathcal{RD}_k^i$ to the actual resource demand rd_k^i of the according software component is done.

3. The defined resource provisionings rp_k^j from each hardware component $hw_j \in \mathcal{HW}$ complete the assignment problem for this configuration.

4. The assignment problem is formalised as a CSP.

5. The CSP is processed by a Commercial Off-The-Shelf (COTS) solver. The goal is to assign boolean values to the previously defined variables for resource-wise assignment (rz) and component-wise assignment (cz).

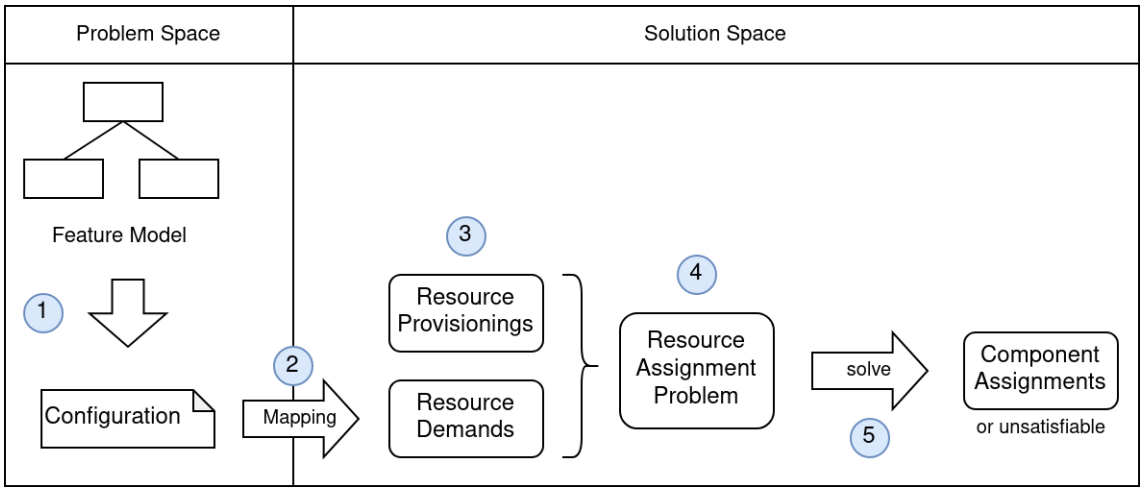

Figure 4.5: The procedure to decide the realisability of a configuration in 5 steps.

Figure 4.2 illustrates the resource demands of the features in the running example. For the configuration $\mathcal{C}_C$ with the features `navigation-system` and `real-time-traffic-system` selected, resource demands to the software components `navigation-system-software` (sw_1) and `rtt-system-software` (sw_2) are generated. They address the resource types RAM (r_0), Response time (r_1) and number of screens (r_2) (see Table 4.1 for details). The following resource demand sets have to be considered: $\mathcal{RD}_0^1 = \{6\}$, $\mathcal{RD}_1^1 = \{50, 30\}$, $\mathcal{RD}_2^1 = \{1\}$, $\mathcal{RD}_0^2 = \{2\}$ and $\mathcal{RD}_1^2 = \{30\}$.

With the reduce function, step 2 yields the actual resource demands of the software components `navigation-system-software` and `rtt-system-software` shown in Table 4.2. The resource provisionings of the hardware components are presented in Table 4.3

	r_0 (RAM [GB])	r_1 (Resp. Time [ms])	r_2 (no. screens)
`navigation-system-software`	6	30	1
`rtt-system-software`	2	30	$\perp$

Table 4.2: Resource demands of the software components `navigation-system-software` and `rtt-system-software` in the running example.

	r_0 (RAM [GB])	r_1 (Resp. Time [ms])	r_2 (no. of screens)
`cu-1`	4	10	$\perp$
`cu-2`	8	10	1

Table 4.3: Resource provisionings of the hardware components `cu-1` and `cu-2` in the running example.

The CSP for the given resource demands and provisionings is formalised and solved. A possible solution is the deployment of the software component `navigation-system-software` to the hardware component `cu-2` and the deployment of the software component `rtt-system-software` to the hardware component `cu-1`. Another solution is the deployment of both software components to the hardware component `cu-2`. Both possible solutions (component assignments) are illustrated in Figure 4.6.

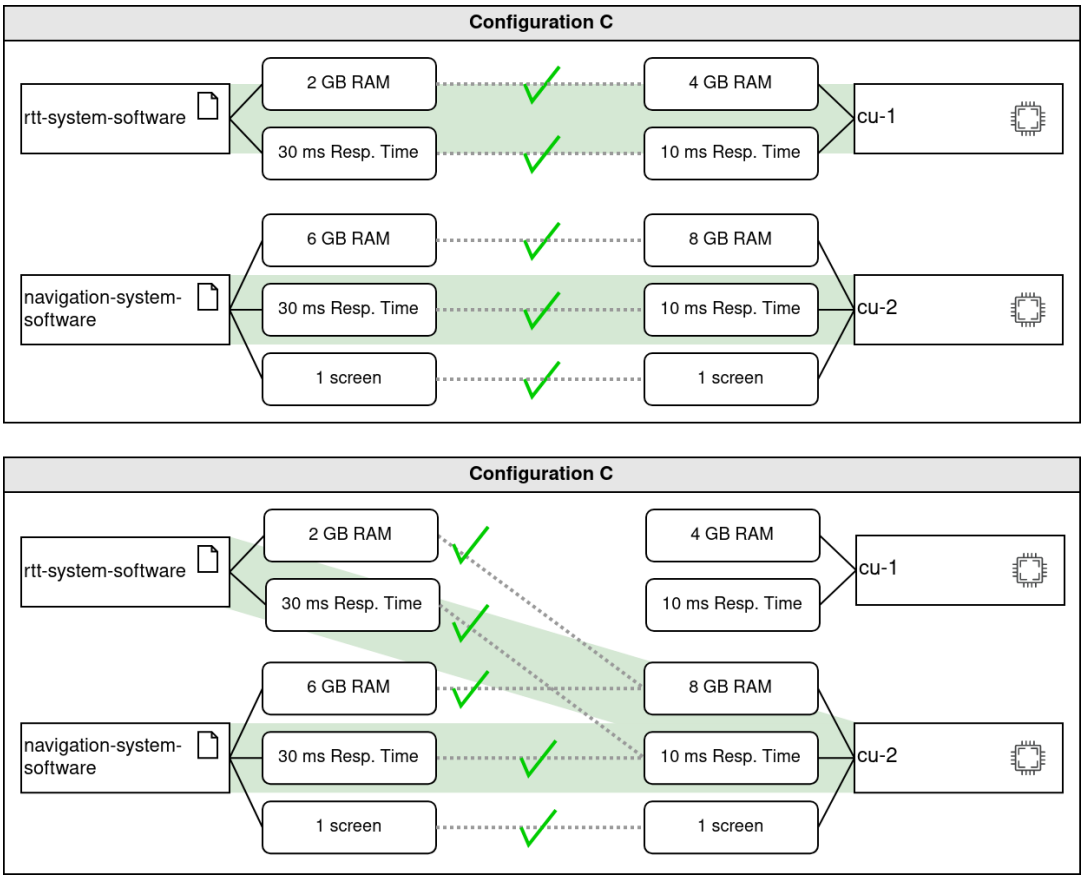

Figure 4.6: Two possible resource assignments (dotted lines) and component assignments (green stripes) for configuration $\mathcal{C}_\mathrm{C}$ of the running example.

4.2.3 Implications of Results

The CSP designed as a decision problem, rather than an optimisation problem, can either be satisfiable or not. The latter case denotes that there is no resource-wise assignment rz from resource demands to resource provisionings found, which satisfies

all constraints. Further, it means that no component-wise assignment cz from software components to hardware components is found, which satisfies all constraints. In this case, it can be said that the configuration in question is not realisable.

In the other case, there is *at least* one variable assignment found which satisfies all constraints. This is called a *satisfiability model*. The model, firstly, expresses all resource-wise assignments rz from resource demands to resource provisionings and, secondly, all component-wise assignment cz from software components to hardware components. For each software component sw_i, the component-wise assignment variable $cz^{i,j}$ to exactly one hardware component hw_j must be evaluated **True** (enforced by Equation 4.7). Deploying this software component to the according hardware component means, that all resource demands of the software component are satisfied by the resource provisionings of the hardware component.

To decide the configuration in question, finding exactly one satisfiability model suffices. As no further selection or optimisation criteria is defined on the satisfiability model, it can theoretically not be influenced, of which character the computed satisfiability model will be. Mostly, it will probably depend on the algorithm used by the solver. Therefore, finding any satisfiability model implies that the configuration in question is realisable.

4.3 Decide the Consistency of Problem and Solution Space

In Section 4.2, the concept to decide the realisability of a single configuration was introduced. To generalise the contributions, the decision of the consistency between problem and solution space is examined now. In case of a given consistency, all valid configurations are also realisable. Two methods are proposed, both building up on the previously introduced concept to decide a single configuration.

4.3.1 Sampling Method

The sampling method is based on the idea to decide a sampled set of valid configurations on their realisability. Sampling configurations is a common practice in problem space analysis, where a representative subset of all valid configurations $\mathcal{CS}_{\text{sampled}} \subseteq \mathcal{CS}$ is generated [HPS+22]. This reduces the number of configurations which have to be regarded in analysis processes.

Depending on the sampling algorithm, a certain coverage of the configuration space $\mathcal{CS}$ is achieved. The sampling method, as proposed here, is defined for the full coverage of the configuration space ($\mathcal{CS}_{\text{sampled}} \equiv_{\mathcal{CS}} \mathcal{CS}$). This way, it is mathematically trivial to decide about the consistency of the problem and solution space, because the resulting implications cover the configuration space completely.

4.3.1.1 Procedure

The procedure is closely related to the one for deciding the realisability of a single configuration (see Listing 4.1 and Subsection 4.2.2). Instead of only a single configuration, the procedure is repeated for every sampled configuration $C \in \mathcal{CS}_{\text{sampled}}$. As the attributed FM stays the same, especially the defined resource types $\mathcal{R}$ and resource provisionings $\mathcal{RP}$ also stay the same for each iteration. Listing 4.2 shows the procedure of the sampling method, referring to the csp-function used in Listing 4.1 to generate a CSP.

```
1  R resource types (input constants)
2  RP resource provisionings (input constants)
3
4  function samplingMethod(CS: set of configurations):
5      isConsistent ← True boolean variable to decide space consistency
6
7      ∀C ∈ CS :
8          compute set of resource demands RD for C via mappings
9          t_p := csp(RD)
10         isConsistent = isConsistent ∧ eval(p)
11
12     return isConsistent
```

Listing 4.2: The sampling method checks, if each valid configuration is also realizable.

4.3.1.2 Implications of Results

As seen in Listing 4.1, the consistency of problem and solution space is given if all (valid) configurations are also realisable. For larger PLs with many valid configurations, the sampling method therefore could be infeasible, as solving a single assignment problem can be already exponential in computational complexity.

While this holds for the consistency, it does not for the *inconsistency*. Diverging problem and solution spaces can be detected by a single configuration that is valid but not realisable. The procedure proposed in Listing 4.1 therefore could be simplified to break after processing the first configuration which is not realisable.

Table 4.4 formally describes the implications. For reasons of completeness, the rightmost column describes the criteria for a generalised sampling algorithm. In this case, certain coverage criteria must be fulfilled first in order to decide the consistency. It enables alternative definitions of space consistency than the one used here.

Decision	Full Coverage ($\mathcal{CS}_{\text{sampled}} \equiv_{\mathcal{CS}} \mathcal{CS}$)	Generalized ($\mathcal{CS}_{\text{sampled}} \subseteq \mathcal{CS}$)
consistency	$\Leftrightarrow \mathcal{RS} \equiv_{\mathcal{CS}} \mathcal{CS}$	$\Leftrightarrow$ coverage criteria fulfilled $\wedge\ \mathcal{RS} \equiv_{\mathcal{CS}} \mathcal{CS}_{\text{sampled}}$
inconsistency	$\Leftrightarrow \mathcal{RS} \subsetneq \mathcal{CS}$	$\Leftrightarrow \mathcal{RS} \subsetneq \mathcal{CS}_{\text{sampled}}$

Table 4.4: Formal criteria to decide consistency and inconsistency between problem and solution space.

4.3.2 Combined Method

The combined method is based on the idea of conjuncting problem space constraints and solution space constraints. A CSP of the problem space is used to generate valid configurations. A CSP of the solution space is used to generate realisable configurations. The method describes, how both CSPs can be combined and then solved in a single step. It addresses the downsides of the sampling method: Mainly its computational complexity and its scattered multi-step approach in practice.

4.3.2.1 Formalising the Unified Conceptual Model as a Constraint Satisfaction Problem

The goal is to formalise a CSP, which expresses both the problem and the solution space as parts of the UCM.

Problem Space

A formalisation of the problem space can be found in the literature and can be seen as a propositional logic representation of a FM. A feature $f \in \mathcal{F}$ is represented by a boolean variable t_f and relates to the select function (see Equation 4.14).

The tree-constraint t_{tree} is adopted from Apel et al. [ABKS13] and was introduced in Table 3.1: All possible feature-dependencies of a FM with the according propositional logic term. The cross-tree constraints are conjuncted to $t_{\text{cross-tree}}$ and then conjuncted with t_{tree}.

The resulting constraint represents the problem space and is denoted with $t_{\mathcal{F}}$. An instantiation of the feature variables t_f along Equation 4.14 represents a configuration.

$$\forall f \in \mathcal{F} : \text{sel}(f) = \text{eval}(t_f) \tag{4.14}$$

Annotations of the FM, e.g. the resource demands of the features, are represented by constants.

Solution Space

The formalisation of the solution space is mainly taken from Subsection 4.2.1 and denoted by t_p. The configuration assumed there is based on the instantiation of feature variables t_f in the CSP of the problem space ($t_{\mathcal{F}}$).

The main difference is, that resource demands are no longer constants, as they are not known in advance, but defined by the mapping. In the combined CSP, the actual resource demand values are defined in the problem space by features. Therefore, the mapped resource demands of software components are represented by numeric variables $t_{rd_k^i}$.

Mappings

Mappings form the connection between the problem space constraint $t_{\mathcal{F}}$ and the solution space constraint t_{p}. First, the problem space constraint defines which features are selected and which are deselected to get a valid configuration. Second, the mapping describes a set of resource demands for the configuration by using the reduce function. Last, the solution space constraint defines the assignment problem to decide the configuration on realisability.

The reduce function was defined in Equation 4.4. For the time of the constraint definition, the actual selection of features is not known. Therefore, a *symbolic function* sym (see Equation 4.15 for a formal definition) of the reduce function is defined in Equation 4.16. It can handle inputs containing logical variables, for example the symbolic set of resource demands of selected features: $\text{sym}(\overline{\mathcal{RD}_k^i}) := \{rd_{k_l}^i \,|\, \text{eval}(t_{f_l}) = \textbf{True}\}$. For the output, the case distinction of the reduce function also returns only symbolic functions (of the sum, the maximum, the minimum etc.), depending on the resource type r_k in question.

The mapping is conjuncted in the constraint t_{red} as illustrated in Listing 4.3

$$\phi(\cdot) \equiv \text{eval}(\text{sym}(\phi)(\cdot)) \tag{4.15}$$

$$\operatorname{sym}\operatorname{red}(\overline{\mathcal{RD}_k^i}) = \begin{cases} \operatorname{sym} \sum_{f_l \in \mathcal{C}} rd_{k_l}^i & \text{if } \mathrm{v}_{\mathrm{add}}(r_k) = \textbf{True} \\ \operatorname{sym}\max(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{LOWER} \\ \operatorname{sym}\min(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{UPPER} \\ \operatorname{sym}\operatorname{fst}(\overline{\mathcal{RD}_k^i}) & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{EXACT} \\ \operatorname{sym} \perp & \text{if } \operatorname{sym}(|\overline{\mathcal{RD}_k^i}|) = 0 \text{ and else} \end{cases} \tag{4.16}$$

```
1  R resource types (input constants)
2  RD resource demands of features (input constants)
3
4  function mappingCSP():
5      T ← ∅ set of constraints
6
7      ∀i ∈ {0, ..., |SW| − 1} :
8          ∀k ∈ {0, ..., |R| − 1} :
9              T ∪ {t_{rd_k^i} ⇔ sym red(RD_k^i)}
10
11     return ⋀(T)
```

Listing 4.3: Assembly of the mapping constraint t_{red} along the symbolic reduce function.

4.3.2.2 Procedure

The combined CSP t_{csp} can be assembled as illustrated in Listing 4.4. Besides the variable declarations, several information is required: First, an annotated FM defines both the problem space constraint $t_{\mathcal{F}}$ and the resource demands from features ($rd_{k_l}^i$). Second, all resource types have to be defined. Third, the resource provisionings of hardware components have to be known. Then, the combined CSP is a conjunction of the problem space constraint $t_{\mathcal{F}}$, the solution space constraint t_{p} and the mapping constraint t_{red} (Listing 4.3).

In comparison to the sampling method, just the single step of solving the combined CSP t_{csp} is necessary to compute configurations both valid and realisable. To compute all such configurations, the procedure must be repeated as long as there are undetected solutions left to the combined CSP. To circumvent the repeated computation of already known satisfiability models, each one has to be added to the set of constraints as an exclusion. Listing 4.5 illustrates the procedure of computing all possible solutions.

4.3.2.3 Implications of Results

If the combined CSP is unsatisfiable, there exists no configuration which is both valid and realisable. Furthermore, no statements about valid configurations could be made, although there might exist some.

If at least one satisfiability model is found for the combined CSP (Listing 4.4), it expresses a configuration both valid and realisable. In detail, the following information can be derived:

```
1  F set of annotated features
2  R set of resource types
3  RP set of resource provisionings
4
5  function combinedCSP():
6      T ← ∅ set of constraints
7
8      t_F := t_cross−tree ∧ t_tree
9      T ∪ {t_F} # problem space
10
11     T ∪ {t_p} # solution space
12
13     T ∪ {t_red} # mappings
14
15     return ⋀(T)
```

Listing 4.4: Assembling the combined CSP with three parts: Problem space, solution space, mappings

```
1  procedure solveCombinedCSP():
2      t_csp′ ← combinedCSP()
3
4      while eval(t_csp′) returns a solution:
5          output solution # satisfiability model
6
7          T_F ← ∅ set of feature variable constraints
8
9          ∀f ∈ F: # collect feature variable assignments
10             T_F ∪ {t_f ≠ eval(t_f)} # collect feature variables
11
12         t_csp′ ∧ ⋁(T_F) # add configuration as exclusion
```

Listing 4.5: Compute all configurations both valid and realisable by repetitive solving and extending of the combined CSP.

1. A configuration $\mathcal{C} \subseteq \mathcal{F}$ is defined by the selected features $f \in \mathcal{F}$. The satisfiability model expresses the configuration with its feature variable assignment $\text{eval}(t_f)$ as described in Equation 4.14.

2. The component assignment for the configuration can be derived from the component assignment variables $cz^{i,j}$, which assign each software component sw_i a hardware component hw_j to be deployed on (see Subsection 4.2.3)

The set of all distinct satisfiability models (see Listing 4.5) expresses all configurations both valid and realisable: $\mathcal{RS}$. As the set of all only-valid configurations $\mathcal{CS}$ is not explicitly computed, the consistency of the problem and solution space can not directly be derived. This also holds for the *inconsistency* of the problem and solution space. Still, the criteria of consistency and inconsistency defined in Table 4.4 is also applicable here: The computed set of configurations both valid and realisable $\mathcal{RS}$ must be compared to the set of only-valid configurations $\mathcal{CS}$.

The main advantage of the combined method is its practical usability, where often only configurations both valid and realisable are of interest. With the sampling

method, all valid configurations $\mathcal{CS}$ have to be computed first before "filtering" them (to $\mathcal{RS}$). With the combined method, the desired set $\mathcal{RS}$ can be computed directly.

For the running example, the attributed FM is shown in Figure 4.2. Formalising the problem space constraint $t_{\mathcal{F}}$ as defined in Table 3.1 yields:

$$
\begin{aligned}
&\texttt{infotainment} \\
\wedge\ &\big((\texttt{radio} \vee \texttt{navigation-system}) \\
&\quad \Leftrightarrow \texttt{infotainment} \wedge \neg(\texttt{radio} \wedge \texttt{navigation-system})\big) \\
\wedge\ &(\texttt{real-time-traffic-system} \Rightarrow \texttt{navigation-system})
\end{aligned}
$$

Formalising and solving the combined problem t_{csp} for the running example expresses three valid and realisable configurations $\mathcal{RS}$: Configuration $\mathcal{C}_{\text{A}}$ (feature `radio`), $\mathcal{C}_{\text{B}}$ (feature `navigation-system`) and $\mathcal{C}_{\text{C}}$ (features `navigation-system` and `real-time-traffic-system`). This yields the set $\mathcal{RS} = \{\mathcal{C}_{\text{A}}, \mathcal{C}_{\text{B}}, \mathcal{C}_{\text{C}}\}$. As these three configurations holistically describe the set of all valid-only configurations $\mathcal{CS} = \{\mathcal{C}_{\text{A}}, \mathcal{C}_{\text{B}}, \mathcal{C}_{\text{C}}\}$, the problem and solution space of the running example can be considered as consistent ($\mathcal{RS} \equiv_{\mathcal{CS}} \mathcal{CS}$).

4.4 Description of Run-Time Variability

The concepts introduced previously address aspects during the build-time of a PL. Therefore, design and management of the PL are typically done in the problem space. For a PL in production and use, modifications can also occur in the solution space only. In the following, a concept called *Capabilities* is proposed. It enables the transformation of solution space modifications into the concept of feature revisions and system revisions. This way, design and management of the PL are shifted back in the problem space and the unidirectional propagation of design decisions from the problem space to the solution space can be maintained. Finally, problems of the run-time variability can be reduced to the contributions of this thesis.

4.4.1 Capabilities

The goal of capabilities is to express solution space modifications. The UCM describes components (class `Component`) and resources (class `Resource`) as central artefacts of the solution space. Therefore, capabilities have to blend into these. Wittler et al. [WKR22] proposed *Services* (class `Service`) in the UCM. The abstract concept can be used to describe dependencies between components, regardless if these "are physical parts [hardware components] or programs [software components]" [WKR22]. In this thesis, services are modified and used as capabilities.

In detail, the following modifications to the UCM from Wittler et al. are made. The resulting UCM is shown in Figure 4.7

1. **Modification**: Renaming `Service` to `Capability`.
2. The class `Capability` has no properties.

   Modification: A property `identifier` (type string) and `metadata` (type `CapabilityMetadata`) is added.

   Intention: A key-value-store should suffice most use cases, as the actual metadata stored is not relevant in the scope of this thesis.
3. The class `Component` has the associations `demands` and `provides` to the class `Capability`.

   Modification: Both associations are specialised to aggregations.

   Intention: Capabilities can exist without components. This principle was also proposed for resources in Section 4.1.

Capabilities can be demanded and provided by software and hardware resources. This is a major difference to the concept of resources, which can only be demanded by software components and provided by hardware components. Particularly, this means that, besides others, software components can have dependencies to other software components. To address these modelling aspects, two demonstrations are presented in the following sections.

Formally, in this thesis the set of capabilities is denoted by $\mathcal{S}$ with a capability $s_m \in \mathcal{S}$ ($\forall m \in \{0, \ldots, |\mathcal{S}| - 1\}$). A component demands and provides a capability s_m as defined in Table 4.5. To keep a modification of the solution space simple, properties

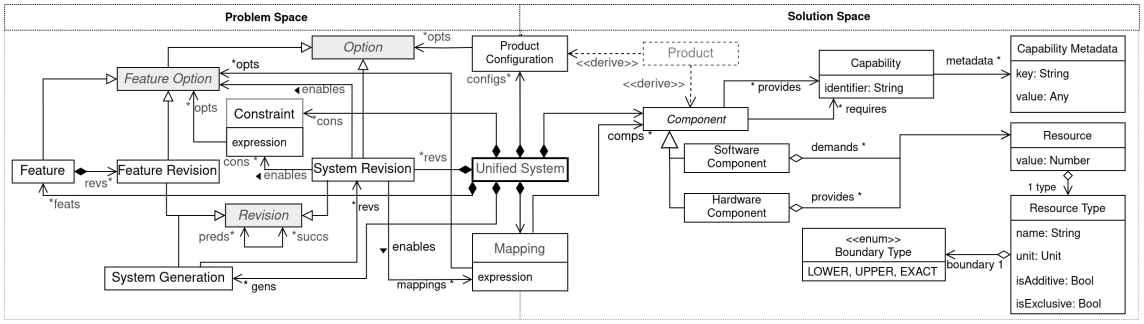

Figure 4.7: The UCM after modifying services to capabilities.

	`demands`	`provides`
software component $sw_i \in \mathcal{SW}$	$sd_m^{\mathcal{SW}_i} \in \mathbb{B}$	$sp_m^{\mathcal{SW}_i} \in \mathbb{B}$
hardware component $hw_j \in \mathcal{HW}$	$sd_m^{\mathcal{HW}_i} \in \mathbb{B}$	$sp_m^{\mathcal{HW}_i} \in \mathbb{B}$

Table 4.5: Denotions for software and hardware components demanding or providing a capability $s_m \in \mathcal{S}$.

of capabilities are only defined via its meta-data and not taken into account when comparing demands and provisionings. Thus, quantification is not of interest and a demand or a provisioning can just be given (**True**) or not (**False**).

Further, Equation 4.17 defines the following sets: First, all software components sw_i which specify any demand to the capability s_m are denoted with $\mathcal{SW}_m^{\text{dem}}$. Analogously denoted are the set of software components which specify a provisioning to capability s_m, $\mathcal{SW}_m^{\text{prov}}$, as well as the hardware component subsets $\mathcal{HW}_m^{\text{dem}}$ and $\mathcal{HW}_m^{\text{prov}}$. Finally, Equation 4.18 defines the set of features, which specify a resource demand to any software component in $\mathcal{SW}_m^{\text{dem}}$ ($\mathcal{F}^{\mathcal{SW}_m^{\text{dem}}}$) or $\mathcal{SW}_m^{\text{prov}}$ ($\mathcal{F}^{\mathcal{SW}_m^{\text{prov}}}$).

$$\begin{aligned}
\mathcal{SW}_m^{\text{dem}} &:= \{sw_i \mid sd_m^{\mathcal{SW}_i} = \textbf{True}\} \subseteq \mathcal{SW} \\
\mathcal{SW}_m^{\text{prov}} &:= \{sw_i \mid sp_m^{\mathcal{SW}_i} = \textbf{True}\} \subseteq \mathcal{SW} \\
\mathcal{HW}_m^{\text{dem}} &:= \{hw_i \mid sd_m^{\mathcal{HW}_i} = \textbf{True}\} \subseteq \mathcal{HW} \\
\mathcal{HW}_m^{\text{prov}} &:= \{hw_i \mid sp_m^{\mathcal{HW}_i} = \textbf{True}\} \subseteq \mathcal{HW}
\end{aligned} \tag{4.17}$$

$$\begin{aligned}
\mathcal{F}^{\mathcal{SW}_m^{\text{dem}}} &:= \bigcup_{sw_i \in \mathcal{SW}_m^{\text{dem}}} \left\{ f_l \mid f_l \in \mathcal{F} : \left(\bigvee_{r_k \in \mathcal{R}} \overline{\text{nil}}(rd_{k_l}^i) \right) = \textbf{True} \right\} \\
\mathcal{F}^{\mathcal{SW}_m^{\text{prov}}} &:= \bigcup_{sw_i \in \mathcal{SW}_m^{\text{prov}}} \left\{ f_l \mid f_l \in \mathcal{F} : \left(\bigvee_{r_k \in \mathcal{R}} \overline{\text{nil}}(rd_{k_l}^i) \right) = \textbf{True} \right\}
\end{aligned} \tag{4.18}$$

4.4.2 Demonstration: New Feature Revision

In this demonstration, the case of a dependency between a software and a hardware component is studied. In detail, it is fixed that a capability $s_{m'} \in \mathcal{S}$ is demanded by a set of software components $\mathcal{SW}_{m'}^{\text{dem}}$ and provided by a set of hardware components $\mathcal{HW}_{m'}^{\text{prov}}$. Both sets can possibly be empty.

In the running example, the following setting is assumed: During the production of the PL, it was discovered that the installed GPS antenna of the new supplier is behaving erratically. The system is designed the way that hardware components only pass through the interface of the GPS antenna. Therefore, the software components that use the interface of the GPS antenna must handle the errors themselves. Hence, the production engineers declared the capability `err-gps-antenna` (s_0), which is demanded by the software component `navigation-system-software` and provided by all connected hardware-components (`cu-1` and `cu-2`).

The described capability can be transformed to problem space constraints in four steps:

1. **Define a new resource type:** A new resource type (instance of the class `ResourceType`) $k' \in \mathcal{R}$ is defined with the properties $\{\mathrm{v}_{\mathrm{add}}(r_{k'}) = \textbf{False}$, $\mathrm{v}_{\mathrm{exc}}(r_{k'}) = \textbf{False}$, $\mathrm{v}_{\mathrm{bnd}}(r_{k'}) = \texttt{EXACT}\}$. The resource type is non-additive, because quantification is not regarded with capabilities. It is non-exclusive, because many software components can demand the capability provided by a hardware component. Finally, the boundary can be specified exactly, as the characteristics for the capability are binary (**False** $\rightarrow$ 0 and **True** $\rightarrow$ 1).

2. **Define resource demands:** All features $f \in \mathcal{F}^{\mathcal{SW}^{\mathrm{dem}}_{m'}}$ are extended with a resource demand to the new resource type k' in context of these software components. Formally:

   $\forall sw_i \in \mathcal{SW}^{\mathrm{dem}}_{m'} : \forall f_l \in \mathcal{F}^{\mathcal{SW}^{\mathrm{dem}}_{m'}} : rd^i_{k'_l} = 1$

   This enforces that, as soon as such a feature is selected, the resource related to the capability is demanded. The modification of these feature attributes can be done with new feature revisions.

3. **Define resource provisionings:** All hardware components which provide the capability are extended with a resource provisioning to the new resource type k':

   $\forall hw_j \in \mathcal{HW}^{\mathrm{prov}}_{m'} : rp^j_{k'} = 1$

4. The capability $s_{m'}$ can be removed. The dependencies are equivalently modeled with resources.

In the running example, first a resource type r_3 with the required properties is instantiated. Only the software component `navigation-system-software` (sw_1) must be regarded: The features `navigation-system` (f_2) and `real-time-traffic-system` (f_3) specify any resource demand and therefore must be extended with $rd^1_{3_2} = 1$ and $rd^1_{3_3} = 1$. Finally, the hardware components `cu-1` (hw_0) and `cu-2` (hw_1) are extended by the resource provisionings $rp^0_3 = 1$ and $rp^1_3 = 1$.

4.4.3 Demonstration: Dependencies between Software Components

In this demonstration, the case of a dependency between software components is studied. In detail, a capability $s_{m'} \in \mathcal{S}$ is demanded by a set of software components $\mathcal{SW}^{\mathrm{dem}}_{m'}$ and provided by a set of software components $\mathcal{SW}^{\mathrm{prov}}_{m'}$. To avoid self-dependencies, it can be required that $\mathcal{SW}^{\mathrm{dem}}_{m'} \cap \mathcal{SW}^{\mathrm{prov}}_{m'} = \emptyset$. Nevertheless, this

requirement is not essential as a self-dependency of a software component is both required and satisfied when the software component is used.

In the running example, the following setting is assumed: During the production of the PL, it was discovered that the software component `radio-software` depends on functionality implemented in the software component `navigation-system-software`. The production engineers then declared the new capability `dep-radio-navigation` (s_1), which is demanded from the software component `radio-software` and provided by the software component `navigation-system-software`.

The described dependencies can be transformed to problem space constraints by defining a new cross-tree constraint for the FM: As soon as any software component from $\mathcal{SW}^{\text{dem}}_{m'}$ is implied by the selection of a feature in $\mathcal{F}^{\mathcal{SW}^{\text{dem}}_{m'}}$, there must be at least one software component in $\mathcal{SW}^{\text{prov}}_{m'}$ implied by any feature in $\mathcal{F}^{\mathcal{SW}^{\text{prov}}_{m'}}$. Formally:

$$\bigvee \left(\mathcal{F}^{\mathcal{SW}^{\text{dem}}_{m'}}\right) \Rightarrow \bigvee \left(\mathcal{F}^{\mathcal{SW}^{\text{prov}}_{m'}}\right)$$

Introducing new cross-tree constraints to the FM can have certain effects. In the running example, the capability `dep-radio-navigation` (s_1) introduces the dependency from the software component `radio-software` (sw_0) to the software component `navigation-system-software` (sw_1). First, the software component `radio-software` is demanded only by the feature `radio` (f_1), implying $\mathcal{F}^{\mathcal{SW}^{\text{dem}}_1} = \{f_1\}$. Next, the software component `navigation-system-software` is only demanded by the feature `navigation-system` (f_2), implying $\mathcal{F}^{\mathcal{SW}^{\text{prov}}_1} = \{f_2\}$. Therefore, the new cross-tree constraint is the feature `radio` implying the feature `navigation-system` ($f_1 \Rightarrow f_2$). But, as these two features are siblings in an alternative-group of the FM, they can not be selected at the same time (referring to Table 3.1). For a customer, this means that the feature `radio` can not be selected anymore.

Dependencies between software components in the solution space is problematic. The idea of software components as an abstraction layer between features and resources enable a free, independent encapsulation of resource demands. To avoid dependencies between software components, the software components itself should be designed the way that they state all their (resource) demands by themselves. It should be discussed, if such dependencies should be resolved in cross-tree constraints or in adaptions of resource demands from features to software components.

4.5 Summary

This thesis addresses the analysis of the solution space of a PL. Configurations which are valid in the problem space are not necessarily realisable in the solution space. Formally, the set of all realisable configurations $\mathcal{RS}$ could only be a subset of all valid configurations $\mathcal{CS}$ ($\mathcal{RS} \subset \mathcal{CS}$). This leads to the divergence of the problem and solution space and impacts further product analysis and testing methods.

To structure the described problems, we stated three research questions (RQ1, RQ2, RQ3) in Chapter 1. The contributions of this thesis to the research questions are discussed in the following.

RQ1: How can solution space artifacts be formally described in a meta-model?

In Section 4.1 we described the solution space of a PL with a meta-model. We modified the UCM, proposed by Wittler et al. [WKR22], along the concepts of components and resources. Software components demand resources while hardware components provide resources. Resource types describe and control the behaviour of resources. In detail, we introduced three properties of a resource type (`isAdditive`, `isExclusive`, `boundary`).

To specify actual resource demand values, we attributed features in a FM with contextual information of the resource demand, such as the addressed software component and the resource type. We argue, that the specification of resource demands can be seen as a task of domain and application engineering and therefore is shifted into the problem space of a PL.

For a concrete configuration, we described the translation from problem space description to solution space artefacts by a mapping between features and software components. We formalised the mapping in the reduce-function. The reduce-function maps a set of resource demands from attributed features to actual resource demand values of a software component, aligning with the UCM which we defined previously as solution space description.

RQ2: How can the realisability of a valid configuration be decided?

In Section 4.2 we defined a CSP to decide the realisability of a configuration. The CSP contains propositional logical representations of solution space artefacts, namely components, resources and resource types. It decides, if all resource demands of the software components are satisfied by the resource provisionings of the hardware components. We can then state information about which software component has to be deployed to which hardware component to yield a functioning product.

In Section 4.3, we generalised the decision about a single configuration to the consistency of problem and solution space. Both spaces are consistent if all valid configurations of the problem space are also realisable in the solution space. We introduced two methods: With the sampling method, we decide the realisability of a pre-calculated sample of valid configurations. If a valid but non-realisable configuration is found, we assume that the problem and solution space diverge. In practice, the sampling method can be used to decide whether an existing configuration can receive a software update (with altered resource demands). With the combined method, we compute configurations both valid and realisable directly by formalising a CSP containing both problem and solution space constraints. In practice, these configurations are of interest for further testing.

RQ3: How can run-time variability be described?

In Section 4.4 we give a description of run-time variability. We extended the proposed UCM to the concept of capabilities. Capabilities enable the description of dependencies between solution space artefacts, namely software and hardware components. We argue that these dependencies should be shifted into the problem space of a PL, as it allows the application of problem space concepts such as the revisioning of features and FMs.

We presented two demonstrations of capabilities: The first one covers dependencies between software and hardware components. Such dependencies can be shifted into the problem space by expressing them as resource demands of attributed features. The second demonstration covers dependencies between software components itself. We argue that the concept of software components, as abstraction between features and resource demands, can be used to design PLs without dependencies between those software components. Therefore, we define a formal translation of such dependencies into cross-tree constraints of FMs.

5. Implementation

This chapter presents the code implementations accompanying the thesis. As this thesis is mainly focused on developing conceptual ideas, the implementation itself is focused on supporting the evaluation of the thesis.

Figure 5.1 shows an overview of the implementations. The three colors indicate categories in which this chapter is divided. Section 5.1 introduces COTS tools (blue). We use the *FeatureIDE* to manage the problem space of a PL and the *Z3 Theorem Prover* to solve CSPs in the solution space. Section 5.2 describes exchange formats (yellow) for common use cases such as storing FMs and configurations in the problem space and resource types, resource demands, resource provisionings and results of CSPs in the solution space. Section 5.3 documents the code artefacts (purple) we implemented. These are a mapper tool (1), implementations of the sampling method (2) and the combined method (3) and a tool which tests the equality of sets of configurations (4), used in the evaluation of this thesis.

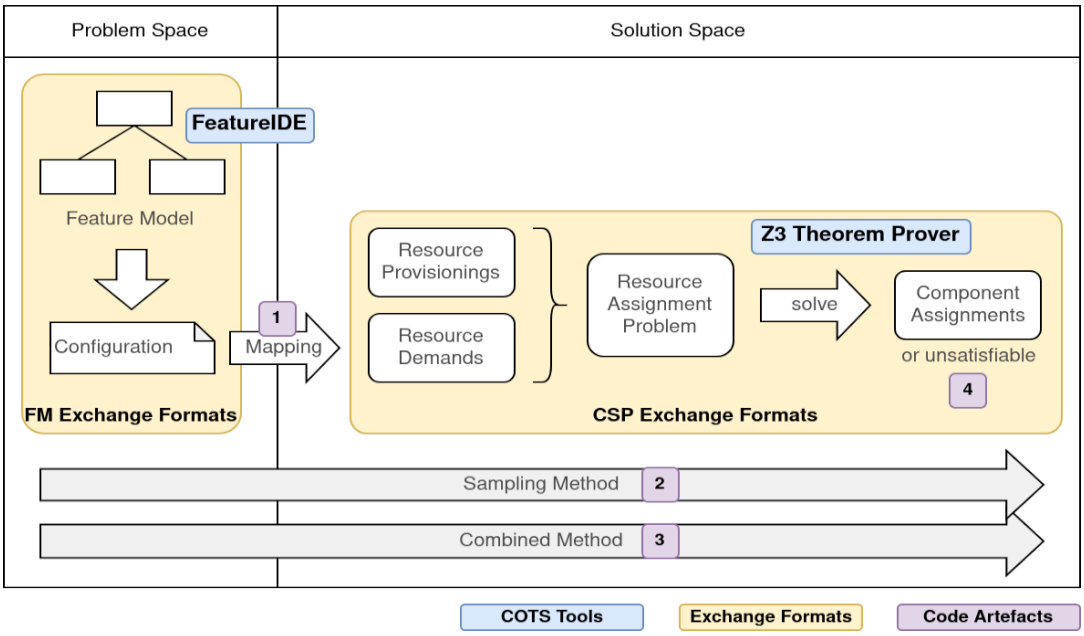

Figure 5.1: Overview of implementations. These are divided into three categories: COTS tools (blue), exchange formats (yellow) and code artefacts (purple).

5.1 COTS Tool Support

This section presents the supporting COTS tools we use: The *FeatureIDE* and the *Z3 Theorem Prover*.

5.1.1 FeatureIDE

We use the *FeatureIDE*[1] [LAMS05] to manage the problem space of a PL. The FeatureIDE is based on the *Eclipse*[2] Integrated Development Environment (IDE) and is a plug-in-solution for feature-oriented software development. The functionality used in this thesis is the modelling and management of FMs, FM dependencies and annotations.

After creating a FM with features and cross-tree constraints, we attributed the features with resource demand descriptions. As proposed in Subsection 4.1.2, such a resource demand (denoted by $rd^i_{k_l}$) is specified in context of a feature $f_l \in \mathcal{F}$, a resource type $r_k \in \mathcal{R}$ and a software component $sw_i \in \mathcal{SW}$. In the FeatureIDE, attributes are realised as a list of key-value items per feature. As shown in Figure 5.2 , we express a resource demand $rd^i_{k_l}$ as an attribute of the feature f_l with the properties $\{\text{key} = (i,k), \text{value} = rd^i_{k_l}\}$. The key of an attribute (type `string`) specifies the context and the value of the attribute (type `long`, an integer) specifies the actual value of the resource demand.

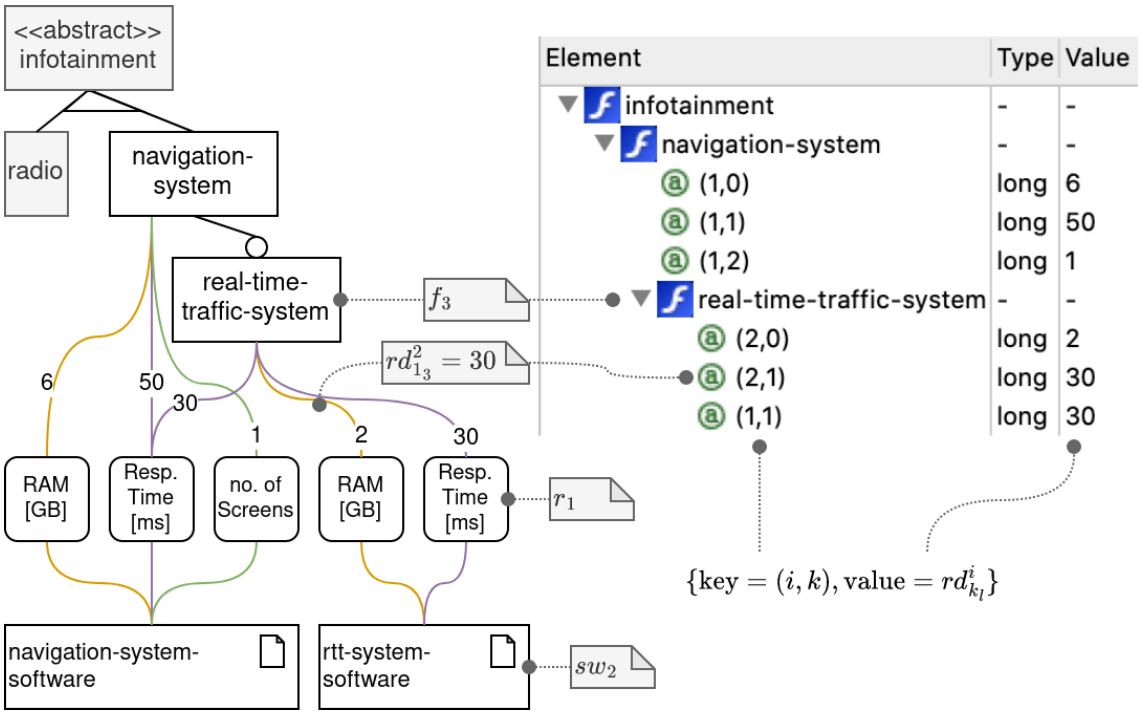

Figure 5.2: In the FeatureIDE, resource demands are specified as feature attributes with properties $\{\text{key}=(i,k),\ \text{value}=rd^i_{k_l}\}$.

[1] https://www.featureide.de/
[2] https://www.eclipse.org/

5.1.2 Z3 Theorem Prover

We use the *Z3 Theorem Prover*[3], initially released in 2012 by Nikolaj Bjorner [Bjø12], to solve CSPs. The CSPs developed in this thesis (for example to decide the realisability of a configuration in Section 4.2) include arithmetic and pseudo-boolean statements, e.g. multiplication and summation. The Z3 Theorem Prover can compute Satisfiability Modulo Theories (SMT) problems, which generalises Boolean Satisfiability Problems (SATs) to express, beside others, real numbers, lists and arithmetics [BH02, CRSC09, DMB11]. Therefore, we assume that the Z3 Theorem Prover is able to compute results for all of our CSPs.

To interact with the Z3 Theorem Prover, Application Programming Interfaces (APIs) for several programming languages are available. For our implementations, we use the Python package `z3-solver`. Section 5.2 describes exchange formats we defined to express the input and output for the tool, for example resource demands, resource provisionings and component assignments. As we separate the data (input and output) from the process (interaction with the Z3 Theorem Prover), the Python implementation should easily be convertible to other programming languages [BMNW21].

[3] https://github.com/Z3Prover/z3

5.2 Exchange Formats

This section presents exchange formats we developed to store data like FMs, configurations and resource demands. Tools rely on these formats as they define input and output interfaces. The section is separated into problem space (Subsection 5.2.1) and solution space (Subsection 5.2.2).

5.2.1 Problem Space

In the problem space, exchange formats for FMs and configurations are discussed.

Feature Model

A FM is structured as a tree with a fixed root feature. Therefore, a tree-like file format like Extensible Markup Language (XML) is typically used to store FMs. The FeatureIDE already has the functionality to serialise a FM to an XML file. Listing 5.1 shows the contents of such a file for the FM of the running example. Feature properties, such as being marked as abstract, are handled as node properties (line 4). The attributes of a feature are handled as child nodes of a feature node. We use the XML serialisation of a FM to process the structure and attributes of a FM.

```
1  <featureModel>
2      [...]
3      struct>
4          <alt abstract="true" mandatory="true" name="infotainment">
5              <feature name="radio">
6                  <attribute name="(0,0)" type="long" value="1"/>
7                  <attribute name="(0,1)" type="long" value="100"/>
8                  <attribute name="(0,2)" type="long" value="1"/>
9              </feature>
10             <and name="navigation-system">
11                 <attribute name="(1,0)" type="long" value="6"/>
12                 <attribute name="(1,1)" type="long" value="50"/>
13                 <attribute name="(1,2)" type="long" value="1"/>
14                 <feature name="real-time-traffic-system">
15                     <attribute name="(2,0)" type="long" value="2"/>
16                     <attribute name="(2,1)" type="long" value="30"/>
17                     <attribute name="(1,1)" type="long" value="30"/>
18                 </feature>
19             </and>
20         </alt>
21     </struct>
22 </featureModel>
```

Listing 5.1: Shortened XML file with the serialised FM of the running example.

Configuration(s)

We use an XML file format to store a configuration. This functionality is already included in the FeatureIDE. Listing 5.2 shows configuration $\mathcal{C}_C$ of the running example where the feature `real-time-traffic-system` was selected manually and the

feature `navigation-system` was selected automatically, because the feature `real-time-traffic-system` depends on the feature `navigation-system`. All features of the FM are serialised in a list. Feature properties, such as their selection or un-selection in the configuration, are expressed as node properties of the feature: Each feature can have the binary properties `automatic` (line 4) or `manual` (line 5), indicating if it is selected automatically (through constraints), manually or not selected (none, line 3).

```
1 <configuration>
2     <feature automatic="selected" name="infotainment"/>
3     <feature name="radio"/>
4     <feature automatic="selected" name="navigation-system"/>
5     <feature manual="selected" name="real-time-traffic-system"/>
6 </configuration>
```

Listing 5.2: XML-serialised configuration $\mathcal{C}_\mathrm{C}$ of the running example. The feature `real-time-traffic-system` was selected manually. Hence, the feature `navigation-system` was selected automatically.

For other applications, such as the combined method, not only one configuration has to be stored but possibly multiple. As each configuration $\mathcal{C} \in \mathcal{CS}$ addresses selected and un-selected features $f \in \mathcal{F}$ (referring to the select-function $\mathrm{sel}(\cdot) \in \mathbb{B}$), a list of configurations has the size $|\mathcal{CS}| \times |\mathcal{F}|$. We store such a generated list of configurations as Comma-Separated Values (CSV) file where each row addresses a configuration and each column addresses a feature. Table 5.1 shows this list for the running example with its three configurations $\mathcal{C}_\mathrm{A}$, $\mathcal{C}_\mathrm{B}$ and $\mathcal{C}_\mathrm{C}$.

configuration	`radio`	`navigation-system`	`real-time-traffic-system`
$\mathcal{C}_\mathrm{A}$	**True**	**False**	**False**
$\mathcal{C}_\mathrm{B}$	**False**	**True**	**False**
$\mathcal{C}_\mathrm{C}$	**False**	**True**	**True**

Table 5.1: List of three configurations $\mathcal{C}_\mathrm{A}$, $\mathcal{C}_\mathrm{B}$, $\mathcal{C}_\mathrm{C}$ for the running example.

5.2.2 Solution Space

In the solution space, exchange formats for resource types, resource demands, resource provisionings and results of solution space CSPs are presented.

Resource Types

Resource types are an essential part of the description of the solution space and therefore important to store. In Section 4.1, we defined the three properties of a resource type (`isAdditive`, `isExclusive`, `boundary`). To encode a resource type, we use the following data set:

1. The identifier (k) is encoded as integer (domain $\mathbb{Z}$).

2. The property `isAdditive` is encoded with an adapted $\mathrm{v}_{\mathrm{add}}$-function:

$$\mathrm{v}_{\mathrm{add}}^{\mathbb{Z}} : \mathcal{R} \to \mathbb{Z}, \quad \mathrm{v}_{\mathrm{add}}^{\mathbb{Z}}(r_k) = \begin{cases} 0 & \text{if } \mathrm{v}_{\mathrm{add}}(r_k) = \textbf{False} \\ 1 & \text{else} \end{cases}$$

3. The property `isExclusive` is encoded with an adapted $\mathrm{v}_{\mathrm{exc}}$-function:

$$\mathrm{v}_{\mathrm{exc}}^{\mathbb{Z}} : \mathcal{R} \to \mathbb{Z}, \quad \mathrm{v}_{\mathrm{exc}}^{\mathbb{Z}}(r_k) = \begin{cases} 0 & \text{if } \mathrm{v}_{\mathrm{exc}}(r_k) = \textbf{False} \\ 1 & \text{else} \end{cases}$$

4. The property `boundary` is encoded with an adapted $\mathrm{v}_{\mathrm{bnd}}$-function:

$$\mathrm{v}_{\mathrm{bnd}}^{\mathbb{Z}} : \mathcal{R} \to \mathbb{Z}, \quad \mathrm{v}_{\mathrm{bnd}}^{\mathbb{Z}}(r_k) = \begin{cases} 0 & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{LOWER} \\ 1 & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{UPPER} \\ 2 & \text{if } \mathrm{v}_{\mathrm{bnd}}(r_k) = \texttt{EXACT} \end{cases}$$

Therefore, a list of all resource types is of two-dimensional shape with size $|\mathcal{R}| \times 4$ (referring to four integers as listed above). We store this data in a CSV file, as it is an easy to use, platform-independent and non-proprietary file format. Figure 5.3 shows the serialisation of the three resource types of the running example.

ID (*k*)	Name	isAdditive	isExclusive	boundary
0	RAM	True	False	LOWER
1	Resp. Time	False	False	UPPER
2	no. of screens	True	False	EXACT

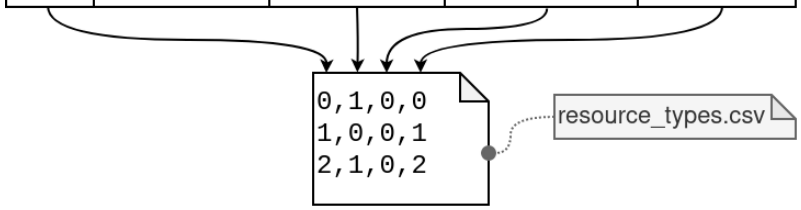

Figure 5.3: Serialisation of resource types in the running example. For three resource types, the serialisation is of size 3×4.

Resource Demands & Resource Provisionings

Resource provisionings of hardware components and resource demands of software components are also an essential part of the description of the solution space. It is necessary to define an exchange format for these data sets, because our sampling method (proposed in Subsection 4.3.1) needs both as input.

Both data sets define values (generally in $\mathbb{R}$) in a two-dimensional context: A resource demand rd_k^i is specified in context of a software component $sw_i \in \mathcal{SW}$ and a

resource type $r_k \in \mathcal{R}$. A resource provisioning rp_k^j is specified in context of a hardware component $hw_j \in \mathcal{HW}$ and a resource type $r_k \in \mathcal{R}$. Therefore, we store all resource demands in a CSV file with size $|\mathcal{SW}| \times |\mathcal{R}|$ and all resource provisionings in a CSV file with size $|\mathcal{HW}| \times |\mathcal{R}|$. Each row ($i$ or j) addresses the component (hardware or software) and each column addresses the resource type (k). Figure 5.4 shows the serialisation of the resource demands for configuration $\mathcal{C}_\mathrm{C}$ of the running example.

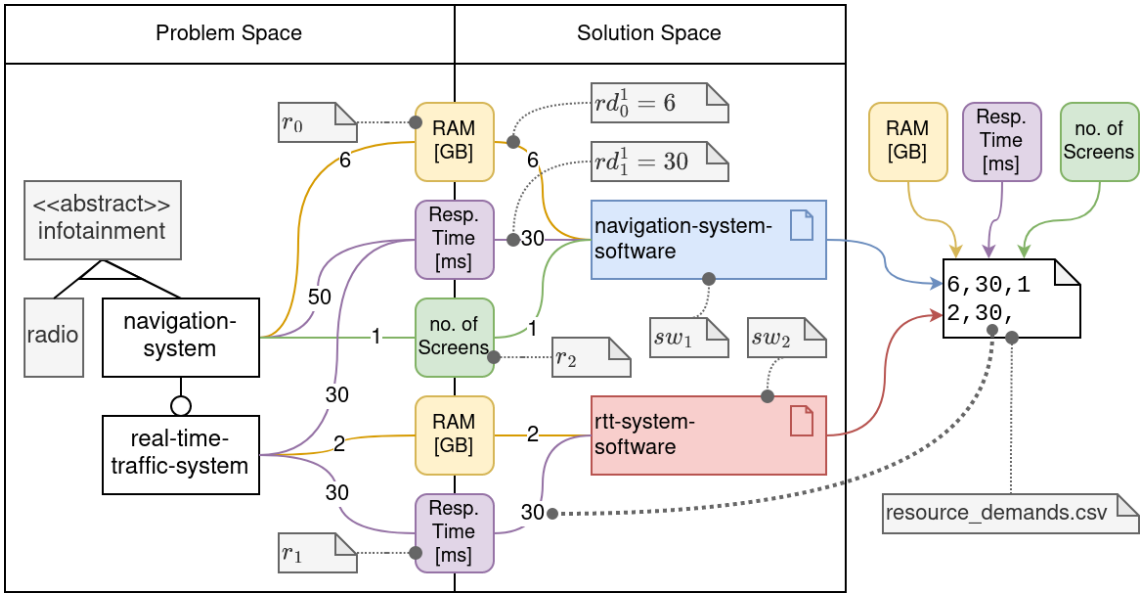

Figure 5.4: Serialisation of resource demands for configuration $\mathcal{C}_\mathrm{C}$ in the running example. For two software components and three resource types, the serialisation is of size 2×3.

CSP Results

In the solution space we formalised the assignment problem t_p in Section 4.2, where software components are assigned to hardware components. We solve this problem with the *Z3 Theorem Prover* described in Section 5.1. As this is the main output of the sampling method, it is important to store the results of the problem for further evaluation. The result can either be a satisfiability model or the information that the CSP is unsatisfiable. In the latter case, any file output can practically be omitted. In the implementation, we write an empty file `unsat` without a file type nonetheless.

If there is a satisfiability model, component assignments exist which assign each software component to exactly one hardware component ($\forall sw_i \in \mathcal{SW} : \forall hw_j \in \mathcal{HW} : cz^{i,j} \in \mathbb{B}$). We store this two-dimensional data as a CSV file, where each row represents a software component and each column a hardware component. Figure 5.5 shows a serialised component assignment for configuration $\mathcal{C}_\mathrm{C}$ of the running example. There is no software component assigned to hardware component `cu-1`. Hence, its column only contains `False`. Both software components are assigned to hardware component `cu-2`. Hence, its column only contains `True`.

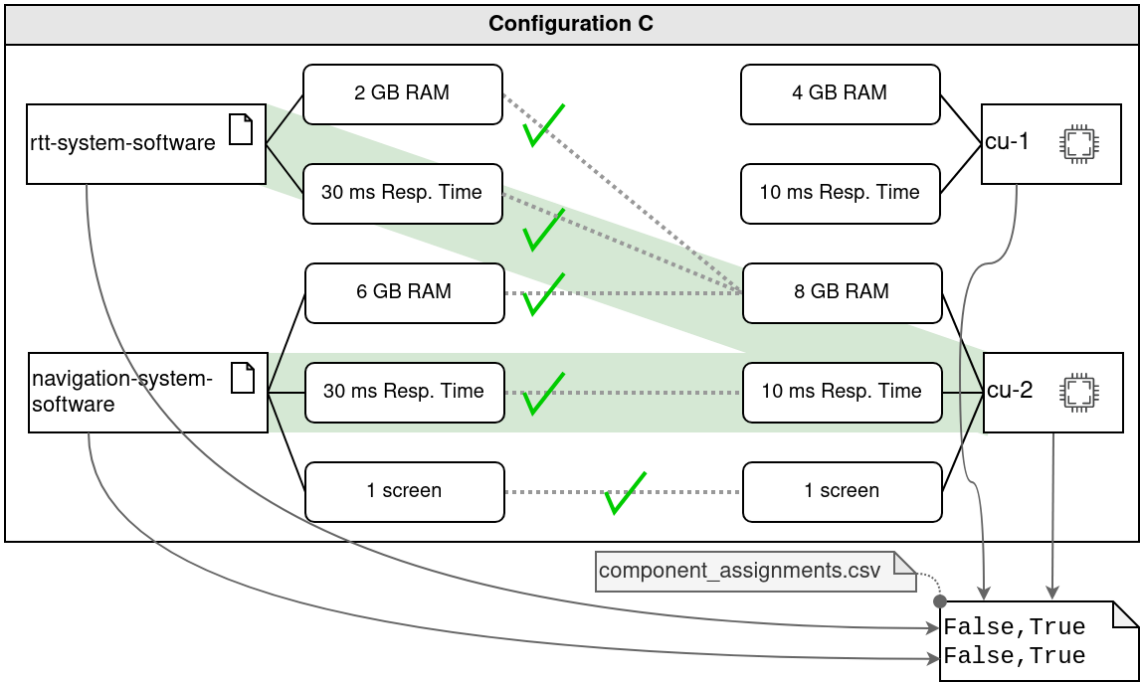

Figure 5.5: Serialisation of component assignments for configuration $\mathcal{C}_{\mathrm{C}}$ in the running example. For two software components and two hardware components, the serialisation is of size 2×2.

5.3 Code Artefacts

This section presents the four code artefacts we implemented and which are shown in Figure 5.1: The mapper tool (Subsection 5.3.1) executes the reduce-function (Equation 4.4) and stores the resource demands of a configuration.

The tool to process the sampling method (Subsection 5.3.2) takes these resource demands, as well as a set of valid configurations, resource provisionings and resource types as input and decides, if the configurations are realisable (referring to the sampling method in Subsection 4.3.1).

The tool to process the combined method (Subsection 5.3.3) takes a feature model and resource provisionings as input and iteratively computes configurations both valid and realisable (referring to the combined method in Subsection 4.3.2).

In the evaluation, the results of the sampling method and the combined method are compared to the expected results. We developed an evaluation tool (Subsection 5.3.4), which decides the result sets for equality ($\equiv_{\mathcal{CS}}$-operator).

5.3.1 Mapper Tool

The mapper tool is a Java project which executes the reduce-function (Equation 4.4) for a set of configurations. Listing 5.3 shows the program in pseudo code.

The tool takes an attributed FM, a definition of the resource types and a list of configurations (line 1 to 3). For each configuration $\mathcal{C} \in \mathcal{CS}$, a set of software components $\mathcal{SW}$ is computed, which are specified by the resource demands of the features (line 7). A two-dimensional list (`swComponentDemandsList`, line 8) is initialised with nil ($\perp$) which will store the resource demands of the software components in context of their resource type with size $|\mathcal{SW}| \times |\mathcal{R}|$.

The list is described in lines 10 to 14 where, for each software component $sw_i \in \mathcal{SW}$ and each resource type $r_k \in \mathcal{R}$, the set of resource demands from features RD_k^i is reduced to a resource demand value rd_k^i.

Omitted in Listing 5.3 is the translation of software component identifiers to row numbers in the resource demands file. In our implementation, this translation is necessary, because not all specified software component identifiers must occur in the resource demands file, for example if the configuration demands resources only for a subset of all software components. As we want to trace the software components later in the component assignments, we need to know which row number in the serialisation corresponds to which software component identifier.

5.3.2 Tool to Process the Sampling Method

The tool is coded in the Python programming language and decides the realisability of a list of sampled configurations. It implements the sampling method proposed in Subsection 4.3.1. Hence, the tool computes the set of all configurations both valid and realisable $\mathcal{RS}$ as a subset of all sampled configurations $\mathcal{CS}_{\text{sampled}}$ ($\mathcal{RS} \subseteq \mathcal{CS}_{\text{sampled}}$).

The resource types and resource provisionings must be given as well as the resource demands (e.g. from the mapper tool) for each configuration. The assembly of the CSP t_{p} and its computation is done with the Z3 Theorem Prover. The result for each configuration is either a file named `unsat`, in case the configuration is not realisable, or a CSV file containing the component assignments.

```
1  F attributed feature model
2  R resource types
3  CS list of configurations
4
5  procedure mapping():
6      ∀C ∈ CS :
7          compute SW
8          init swComponentDemandsList: int[][] ← ⊥
9
10         ∀ sw_i ∈ SW :
11             ∀ r_k ∈ R :
12                 compute RD_k^i
13                 compute rd_k^i = red(RD_k^i)
14                 write swComponentDemandsList[i][k] = rd_k^i
15
16         write resource demands file for configuration C
```

Listing 5.3: The mapper tool executes the reduce-function for a list of configurations.

5.3.3 Tool to Process the Combined Method

The tool is coded in the Python programming language and computes configurations of a PL both valid and realisable. It implements the combined method, proposed in Subsection 4.3.2, by assembling a CSP that describes both solution and problem space constraints at the same time (i.e., t_{csp}).

As inputs, the attributed FM (XML file), the defined resource types and resource provisionings must be given. The assembly of the CSP and its computation is done with the Z3 Theorem Prover. The tool iteratively computes a configuration both valid and realisable and adds the configuration as exclusion constraint to the problem. This loop breaks as soon as there is no more distinct solution for the problem. The output is a list of found configurations and the component assignments for each configuration.

For large PLs, the procedure may not be able to compute *all* configurations both valid and realisable due to computational complexity. Therefore, further criteria to break the loop, for example the maximum execution time, can be implemented.

5.3.4 Evaluation Tool for Problem and Solution Space Consistency

Both the sampling method and the combined method compute a set of configurations which are both valid and realisable. For the evaluation, we are interested in the equality of those sets to the expected set (ground truth) $\mathcal{RS}^{\text{GT}}$. Therefore, we developed a tool which decides the equality of two sets of configurations ($\mathcal{CS}_{\text{B}}$ and $\mathcal{CS}_{\text{A}}$), based on the $\equiv_{\mathcal{CS}}$-operator defined in Chapter 4.

Listing 5.4 shows the procedure of the tool. Both sets $\mathcal{CS}_{\text{A}}$ and $\mathcal{CS}_{\text{B}}$ are given as lists of configurations, where each list entry represents a single configuration and specifies which features are selected for this configuration (Table 5.1). In line 2, the size equality of $\mathcal{CS}_{\text{A}}$ and $\mathcal{CS}_{\text{B}}$ is asserted. If the size equality is not given, the sets can not be equal. Line 4 and 5 define the family of sets of matching configurations $\mathcal{M}^{\text{A}}$. For each $i \in \{0, \ldots, |\mathcal{CS}_{\text{A}}| - 1\}$, the set $\text{M}_i^{\text{A}} \in \mathcal{M}^{\text{A}}$ contains all configurations from $\mathcal{CS}_{\text{B}}$ which are equal to the specific configuration $\mathcal{C}_i^{\text{A}}$. Each set is initially empty (line

```
1  procedure checkEquality(CS_A, CS_B):
2      assert(|CS_A| = |CS_B|)
3
4      ∀i ∈ {0, ..., |CS_A| − 1} : M_i^A ← ∅
5      M^A := {M_0^A, ..., M_{|CS_A|−1}^A}
6
7      ∀C_i^A ∈ CS_A :
8          ∀C_j^B ∈ CS_B :
9              if C_i^A ≡_C C_j^B : M_i^A ∪ {C_j^B}
10
11     ∀M_i^A ∈ M^A : assert(|M_i^A| = 1)
```

Listing 5.4: Assertion of set equality between $\mathcal{CS}_\mathrm{A}$ and $\mathcal{CS}_\mathrm{B}$.

5) and extended in lines 7 to 9. The assertion in line 11 checks, if each configuration in $\mathcal{CS}_\mathrm{A}$ has *exactly one* counterpart in $\mathcal{CS}_\mathrm{B}$. Together with the assertion in line 2, this implies that *every* configuration in $\mathcal{CS}_\mathrm{A}$ has its counterpart in $\mathcal{CS}_\mathrm{B}$. Then, we assume both sets to be equal.

6. Evaluation

This chapter describes the evaluation of the concepts introduced in Chapter 4. The evaluation is focused on a quantitative assessment of the proposed procedures. Mainly, this covers the research questions RQ1 and RQ2, where we contributed a meta-model of the solution space as well as two methods (sampling method and combined method) to decide the consistency of the problem and solution space of a PL.

Setup & Design

We evaluate the methods along two case studies, which are introduced in Section 6.1. The first case study, CS1, is used to maximize the coverage of the developed logical formulas for the CSPs. The second case study, Body Comfort System (BCS), is used to apply our contributions to an already existing example.

For both case studies, we know a Ground Truth (GT) in advance. The GT contains the set of configurations which are both valid and realisable, denoted by $\mathcal{RS}_{\text{CS1}}^{\text{GT}}$ and $\mathcal{RS}_{\text{BCS}}^{\text{GT}}$. We compare the GT against the computed sets of configurations which are both valid and realisable, $\mathcal{RS}_{\text{CS1}}$ and $\mathcal{RS}_{\text{BCS}}$. Section 6.2 describes the evaluation procedure in detail.

In Section 6.3, we present the results as well as the limitations of the evaluation. In Section 6.4 we discuss the evaluation in context of the research questions and goals of this thesis. Threats to the validity of our work are assessed in Section 6.5.

6.1 Case Studies

This section describes the two case studies in detail. For both case studies, the problem and solution space artefacts are introduced and the GT (sets of configurations) is derived.

6.1.1 Case Study 1 (CS1)

Case Study 1 (CS1) is a small example of a car PL and used to evaluate the developed procedure of deciding if a configuration is realisable. In Section 4.2 we presented an approach to compute this decision by constructing a CSP. It consists of a set of propositional logical constraints. To verify these constraints, we designed CS1 the way that it covers all of the proposed logical formulas. As the formulas depend highly on the properties of the resource types (`isAdditive`, `isExclusive`, `boundary`) which occur in the data (e.g. Equation 4.10 to Equation 4.13), CS1 covers all twelve possible characteristics of resource type properties (shown as tree in Figure 4.4).

Feature Model

In the problem space, we defined a FM as shown in Figure 6.1. It represents a PL of a car manufacturer with 13 features (three abstract features) and no cross-tree constraints. The customer can choose between two infotainment systems. Further, the optional motorisation of the seats can either be only in the front row or for all passengers (front and back) and the adjustment of the seats can either be six-way or ten-way. Optionally, the car can be eqipped with a tyre pressure monitoring system. The monitoring system for critical components (CCM) of the car (for example oil pressure and temperature) is mandatory.

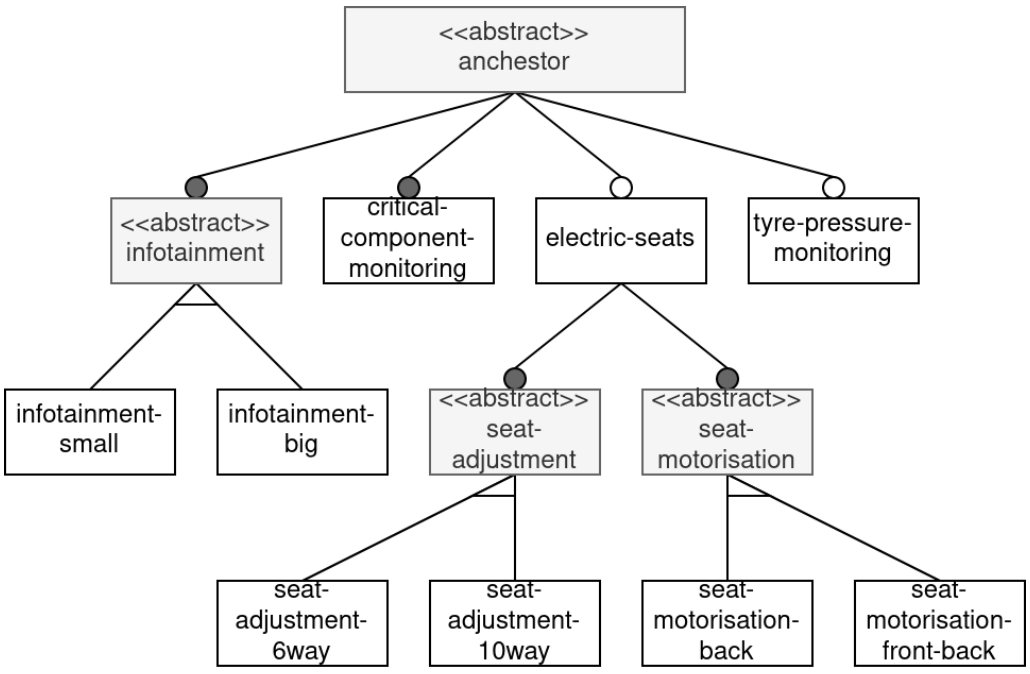

Figure 6.1: FM of CS1.

Resource Types

The intention behind the defined resource types is to cover all twelve characteristics of resource type properties (`isAdditive` $\in \mathbb{B}$, `isExclusive` $\in \mathbb{B}$, `boundary` $\in \{$`LOWER`, `UPPER`, `EXACT`$\}$). Therefore, we specified *exactly twelve distinct* resource types as shown in Table 6.1.

ID (k)	$\mathrm{v}_{\mathrm{add}}(r_k)$	$\mathrm{v}_{\mathrm{exc}}(r_k)$	$\mathrm{v}_{\mathrm{bnd}}(r_k)$	Unit	Description
0	**False**	**False**	`LOWER`	MBit/s	Communication Bandwidth
1	**False**	**False**	`UPPER`	ms	Response Time
2	**False**	**False**	`EXACT`	GHz	Infotainment Core Clock
3	**False**	**True**	`LOWER`	Mbit/s	CCM Communication Bandwidth
4	**False**	**True**	`UPPER`		CCM Screen Resolution
5	**False**	**True**	`EXACT`		Screen Resolution
6	**True**	**False**	`LOWER`	GByte	Memory
7	**True**	**False**	`UPPER`		no. Touchscreens
8	**True**	**False**	`EXACT`		no. Screens
9	**True**	**True**	`LOWER`		no. Seat Adjustment Actuators
10	**True**	**True**	`UPPER`		no. Seats
11	**True**	**True**	`EXACT`		no. Tyre Pressure Sensors

Table 6.1: Resource types for CS1. Screen resolutions are given in a multiple of the generic "1K"-resolution, e.g. 2K (approx. 2000 pixels width), 4K (approx. 4000 pixels width) etc.

Software Components & Hardware Components

We specified five software components and three hardware components as solution space artefacts. They are described in the following.

Table 6.2 shows the five different software components, which relate loosely to the FM. The software component `infotainment-software` (sw_0) addresses the infotainment features. The critical component monitoring is split into a software component for visualisation (`ccm-visualisation-software`, sw_1) and a software component for calculation (`ccm-calculation-software`, sw_2). The seat adjustment is realised as a user interface panel which is controlled by the software component `seat-adjustment-panel-software` (sw_3). The tyre pressures are monitored by the software component `tyre-pressure-monitoring-software` (sw_4).

We specified three different hardware components: The hardware component `infotainment-hardware` (hw_0) for infotainment and communication, `ccm-hardware` (hw_1) for critical components and `car-periphery-hardware` (hw_2) for peripheral components.

Resource Demands and Provisionings

Resource demands are described in the problem space as feature attributes. Table 6.3 shows an excerpt of all specified resource demands in context of a feature f_l, a software component sw_i and a resource type r_k. The complete table is listed in Table A.1. Table 6.4 shows all resource provisionings in context of a hardware component hw_j and a resource type r_k.

ID(i)	Name	Relates to Features
0	`infotainment-software`	`infotainment`
1	`ccm-visualisation-software`	`critical-component-monitoring`
2	`ccm-calculation-software`	`critical-component-monitoring`
3	`seat-adjustment-panel-software`	`electric-seats`
4	`tyre-pressure-monitoring-software`	`tyre-pressure-monitoring`

Table 6.2: Software components for CS1.

Feature	Software Comp. (sw_i)	Demand (rd^i_k)
`infotainment-small`	sw_0	$rd^0_0 = 5$ $rd^0_1 = 100$ $rd^0_2 = 1$ $rd^0_6 = 2$ $rd^0_7 = 1$
`infotainment-big`	sw_0	$rd^0_0 = 10$ $rd^0_1 = 100$ $rd^0_2 = 3$ $rd^0_6 = 6$ $rd^0_7 = 3$
`critical-component-monitoring`	sw_1	$rd^1_4 = 1$ $rd^1_6 = 2$ $rd^1_8 = 1$
	sw_2	$rd^2_3 = 2$ $rd^2_6 = 1$
`electric-seats`	sw_3	$rd^3_5 = 1$
`seat-adjustment-6way`	sw_3	$rd^3_9 = 6$
`seat-adjustment-10way`	sw_3	$rd^3_9 = 10$
`seat-motorisation-front`	sw_3	$rd^3_{10} = 2$
`seat-motorisation-front-back`	sw_3	$rd^3_{10} = 4$
`tyre-pressure-monitoring`	sw_4	$rd^4_{11} = 4$

Table 6.3: Resource demands of features in CS1 (excerpt; complete table in Table A.1.

Hardware Component (hw_j)	Provisioning (rp^j_k)
`infotainment-hardware` (hw_0)	$rp^0_0 = 15$ $rp^0_1 = 50$ $rp^0_2 = 3$ $rp^0_6 = 8$ $rp^0_7 = 1$
`ccm-hardware` (hw_1)	$rp^1_3 = 5$ $rp^1_4 = 1$ $rp^1_6 = 4$ $rp^1_8 = 1$
`car-periphery-hardware` (hw_2)	$rp^2_5 = 1$ $rp^2_9 = 6$ $rp^2_{10} = 2$ $rp^2_{11} = 4$

Table 6.4: Resource provisionings of hardware components in CS1.

Ground Truth

The GT gives prior information about the validity (problem space) and realisability (solution space) of configurations. We computed the set of all valid configurations $\mathcal{CS}_{\text{CS1}}$ with the *FeatureIDE* (Section 5.1) and got the set size $|\mathcal{CS}_{\text{CS1}}| = 20$.

Regarding the realisability, we designed resource demands and resource provisionings that way, that certain configurations are not realisable. This is done by specifying features which yield resource demands that can not be satisfied by the resource provisionings. In detail, we defined the following incompatibilities for CS1:

- The feature `infotainment-small` specifies a resource demand to software component `infotainment-software` (sw_0) of resource type `Infotainment Core Clock` (r_2) with value $rd^0_2 = 1$ (GHz). This resource type has the `boundary` property `EXACT` ($\text{v}_{\text{bnd}}(r_2) = \texttt{EXACT}$).

  There is no hardware component that specifies a resource provisioning for r_2 with *exactly* 1 GHz. Thus, the resource demand of the feature `infotainment-small` can not be satisfied.

- The feature `seat-adjustment-10way` specifies a resource demand to software component `seat-adjustment-panel-software` (sw_3) of resource type `no. Seat Adjustment Actuators` (r_9) with value $rd^3_9 = 10$. This resource type has the `boundary` property `LOWER` ($\text{v}_{\text{bnd}}(r_9) = \texttt{LOWER}$).

  There is no hardware component that specifies a resource provisioning for r_9 with *at least* value 10. Meaningfully, no seat in the car has at least ten seat adjustment actuators. Thus, the resource demand of the feature `seat-adjustment-10way` can not be satisfied.

- The feature `seat-motorisation-front-back` specifies a resource demand to software component `seat-adjustment-panel-software` (sw_3) of resource type `no. Seats` (r_{10}) with value $rd^3_{10} = 4$. This resource type has the `boundary` property `UPPER` ($\text{v}_{\text{bnd}}(r_{10}) = \texttt{UPPER}$).

There is no hardware component that specifies a resource provisioning for r_{10} with *at most* value 4. Meaningfully, no car has four motorised seats installed. Thus, the resource demand of the feature `seat-motorisation-front-back` can not be satisfied.

As each incompatibility concerns one feature and has no dependencies to other incompatibilities, at least one of these features has to be selected to yield a non-realisable configuration. We proposed the CSP t_p in Section 4.2 to decide the realisability of a configuration. To derive a GT for CS1, we state the following implication:

$$\begin{aligned} &\texttt{infotainment-small} \\ \vee\ &\texttt{seat-adjustment-10way} \\ \vee\ &\texttt{seat-motorisation-front-back} \\ \Rightarrow\ &\mathrm{eval}(t_\mathrm{p}) \stackrel{!}{=} \textbf{False} \end{aligned}$$

The expected set of all configurations both valid and realisable $\mathcal{RS}_{\mathrm{CS1}}^{\mathrm{GT}}$ therefore contains all valid configurations except those fulfilling the premise of the implication above. The set size is $|\mathcal{RS}_{\mathrm{CS1}}^{\mathrm{GT}}| = 4$.

6.1.2 Body Comfort System (BCS) Case Study

The BCS Case Study is known in the research topic of Software Product Line Engineering (SPLE). The version used in this thesis was published from Lity et al. [SRMI12] in 2012. It was developed with industry partners from the automotive industry and represents an excerpt of a car PL. We use the BCS case study to apply the proposed concepts to an existing example of a larger PL. While problem space artefacts, such as a FM and configurations, are given, we have to extend the case study with solution space artefacts which fit into the UCM.

Feature Model

In the problem space, the FM consists of 27 features (seven abstract features) and six cross-tree constraints. It can be divided into three sub-trees: Figure 6.2 shows the first sub-tree `hmi` concerning a user interface. Six warning and signal lights (features `led-...`) can be chosen in an optional or-group. Figure 6.3 shows the second sub-tree `doors` concerning convenience and safety functionality such as power windows, heatable mirrors and finger protection. Figure 6.4 shows the third sub-tree `security` concerning optional functionality such as a remote-control key and an alarm system. The cross-tree constraints are as follows:

$$\begin{aligned} &(\texttt{led-alarm-system} \Rightarrow \texttt{alarm-system}) \\ \wedge\ &(\texttt{led-central-locking} \Rightarrow \texttt{central-locking}) \\ \wedge\ &(\texttt{led-heatable} \Rightarrow \texttt{exterior-mirror-heatable}) \\ \wedge\ &(\texttt{power-window-manual} \Leftrightarrow \neg\texttt{rc-key-automatic-power-window}) \\ \wedge\ &(\texttt{rc-key} \Rightarrow \texttt{central-locking}) \\ \wedge\ &(\texttt{rc-key-alarm-system} \Rightarrow \texttt{alarm-system}) \end{aligned}$$

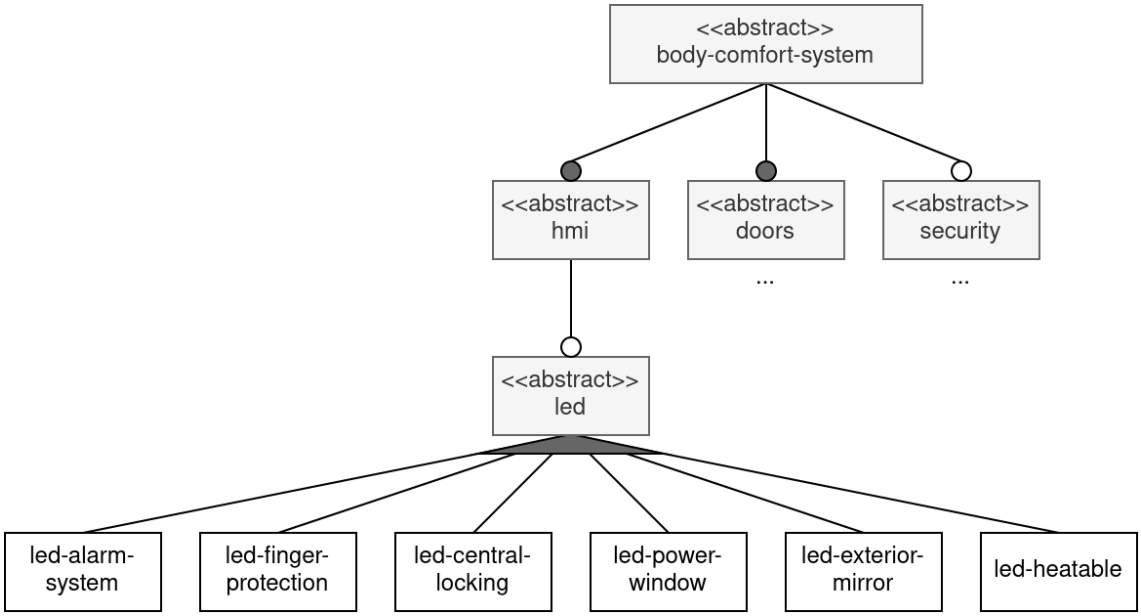

Figure 6.2: Sub-tree `hmi` of the FM of the BCS case study.

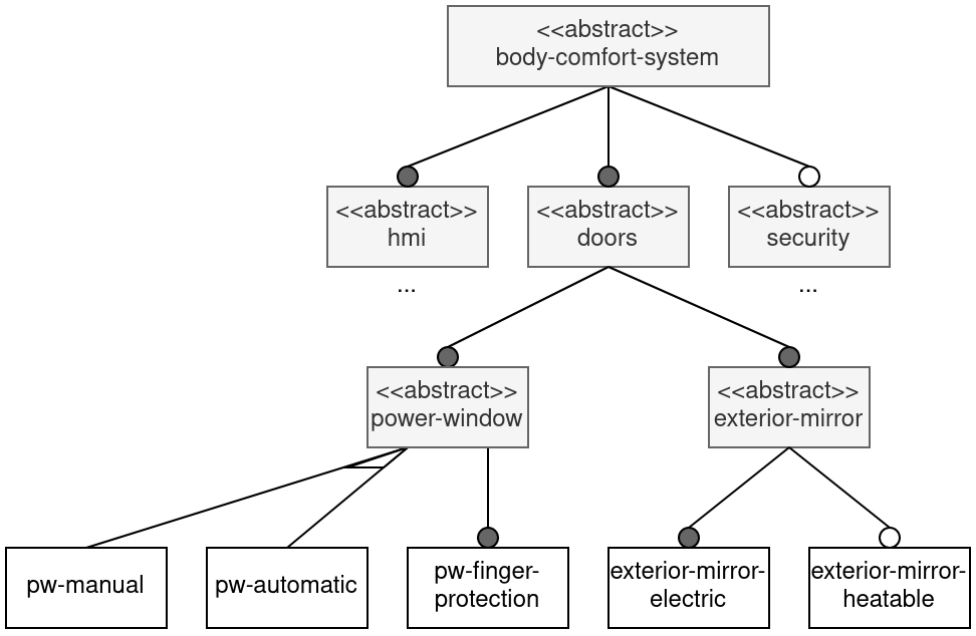

Figure 6.3: Sub-tree `doors` of the FM of the BCS case study.

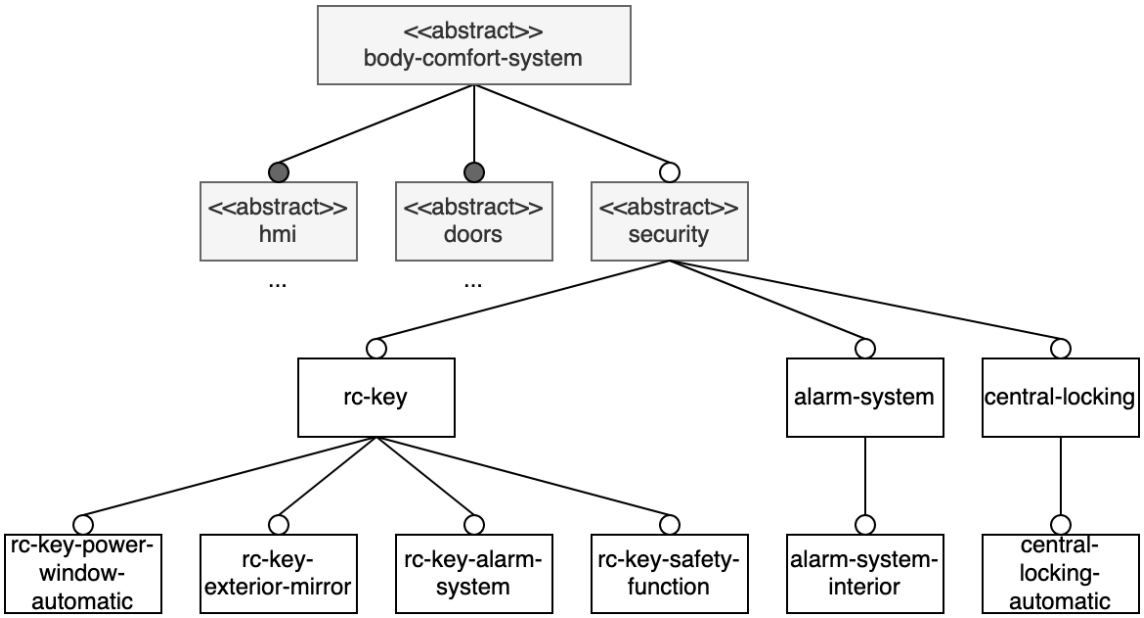

Figure 6.4: Sub-tree `security` of the FM of the BCS case study.

Resource Types, Software Components & Hardware Components

In the solution space, we specified four software components: The software component `hmi-software` (sw_0) encapsulates the implementation of the user interface. The software components `power-window-control-software` (sw_1) and `exterior-mirror-control-software` (sw_2) address the operation of exterior mirrors and power windows. The software component `security-software` (sw_3) controls all security installations in the car.

The resource types are shown in Table 6.5 and address requested resources of the described functionality such as power specifications, security video cameras and communication bandwidth.

Further, we defined two hardware components: The hardware component `security-hardware` (hw_0) encapsulates critical tasks for the security of the car. The hardware component `infotainment-hardware` (hw_1) functions a central CU for all other, non-critical tasks.

ID(k)	$\mathrm{v}_{\mathrm{add}}(r_k)$	$\mathrm{v}_{\mathrm{exc}}(r_k)$	$\mathrm{v}_{\mathrm{bnd}}(r_k)$	Unit	Description
0	**True**	**False**	`LOWER`		no. Interface Slots
1	**True**	**False**	`LOWER`	W	Power Specification
2	**True**	**True**	`LOWER`		no. Security Comm. Channels
3	**False**	**True**	`LOWER`	MHz	Security Processing Core Clock
4	**False**	**True**	`UPPER`	min	Security Automatic Relock Time
5	**True**	**True**	`EXACT`		no. Security Video Cameras
6	**True**	**False**	`LOWER`		no. Window Movement Sensors
7	**True**	**False**	`LOWER`	kbit/s	Bandwidth

Table 6.5: Resource types of the BCS case study.

Resource Demands & Provisionings

Table 6.6 shows an excerpt of the resource demands of the features in context of the four software components ($sw_0, \ldots, sw_3$) and the eight resource types ($r_0, \ldots, r_7$). The complete table is listed in Table A.2. Table 6.7 shows the resource provisionings of the two hardware components (hw_0, hw_1).

Feature	Software Comp. (sw_i)	Demands (rd^i_k)
`led-alarm-system`	sw_0	$rd^0_0 = 2$
`led-finger-protection`	sw_0	$rd^0_0 = 1$
`led-central-locking`	sw_0	$rd^0_0 = 1$
`led-power-window`	sw_0	$rd^0_0 = 1$
`led-exterior-mirror`	sw_0	$rd^0_0 = 1$
`led-em-heatable`	sw_0	$rd^0_0 = 1$
`em-electric`	sw_1	$rd^1_7 = 10$ $rd^1_1 = 5$
`em-heatable`	sw_1	$rd^1_1 = 20$
`pw-finger-protection`	sw_2	$rd^2_6 = 1$
`pw-manual`	sw_2	$rd^2_6 = 2$
`pw-automatic`	sw_2	$rd^2_7 = 5$ $rd^2_6 = 3$
`rc-key`	sw_3	$rd^3_2 = 2$ $rd^3_3 = 10$
`rck-pw-automatic`	sw_2	$rd^2_7 = 5$
	sw_3	$rd^3_2 = 2$
`rck-exterior-mirror`	sw_1	$rd^1_7 = 10$
	sw_3	$rd^3_2 = 2$
`rck-alarm-system`	sw_3	$rd^3_2 = 2$
`rck-safety-function`	sw_3	$rd^3_2 = 2$
`alarm-system`	sw_3	$rd^3_3 = 100$ $rd^3_5 = 4$
`alarm-system-interior`	sw_3	$rd^3_3 = 700$ $rd^3_5 = 1$
`central-locking`	sw_3	$rd^3_3 = 10$
`central-locking-automatic`	sw_3	$rd^3_4 = 1$

Table 6.6: Resource demands of features in the BCS case study (excerpt; complete table in Table A.2).

Ground Truth

The GT for the BCS case study is derived analogously to the CS1. In the problem space, we computed the set of valid configurations $\mathcal{CS}_{\text{BCS}}$ with the *FeatureIDE*. The set is of size $|\mathcal{CS}_{\text{BCS}}| = 7512$.

For the realisability, we designed the resource demands and provisionings analogously to CS1, where some features yield unsatisfiable resource demands. In the

Hardware Component (hw_j)	Provisionings (rp_k^j)
`infotainment-hardware` (hw_0)	$rp_0^0 = 8$ $rp_1^0 = 25$ $rp_6^0 = 4$ $rp_7^0 = 16$
`security-hardware` (hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$

Table 6.7: Resource provisionings of hardware components in the BCS case study.

BCS case study, only the feature `alarm-system-interior` yields an incompatibility. It specifies a resource demand to software component `security-software` (sw_3) of resource type `no. Security Video Cameras` (r_5) with the value $rd_5^3 = 1$. This resource type has the `boundary` property `EXACT` ($\mathrm{v}_{\mathrm{bnd}}(r_5) = \texttt{EXACT}$). There is no hardware component that provides this resource type r_5 with the exact value 1, which means no hardware component provides *exactly one* security camera, which the feature `alarm-system-interior` could use. Thus, the resource demand of this feature can not be satisfied.

The feature `alarm-system-interior` is the only feature that specifies unsatisfiable resource demands. Therefore, we can state the following implication on the CSP of the solution space of the BCS case study (t_{p}):

$$\texttt{alarm-system-interior} \Rightarrow \mathrm{eval}(t_{\mathrm{p}}) \stackrel{!}{=} \textbf{False}$$

The expected set of all configurations both valid and realisable $\mathcal{RS}_{\mathrm{BCS}}^{\mathrm{GT}}$ therefore contains all valid configurations except those containing the feature `alarm-system-interior`. The set size is $|\mathcal{RS}_{\mathrm{BCS}}^{\mathrm{GT}}| = 4200$.

6.2 Evaluation Procedure

This section describes the procedure we followed to evaluate our contributions. Subsection 6.2.1 introduces the evaluation procedure for the sampling method and Subsection 6.2.2 for the combined method.

For both case studies, CS1 (Subsection 6.1.1) and the BCS case study (Subsection 6.1.2), the procedure is the same. As a prerequisite, we aim to achieve full coverage of the configuration space for both case studies: The sampling method is done for *all* valid configurations ($\mathcal{CS}_{\text{CS1}}$ and $\mathcal{CS}_{\text{BCS}}$) and the combined method stops when no more solutions are possible, e.g. *all* configurations both valid and realisable are found.

6.2.1 Sampling Method

For the sampling method, the procedure proposed in Subsection 4.3.1 is followed and shown in Figure 6.5. With the *FeatureIDE*, we generate all valid configurations from the FMs of the case studies, denoted by $\mathcal{CS}_{\text{CS1}}$ and $\mathcal{CS}_{\text{BCS}}$. The resource demands are specified in the attributed FM, following Table 6.3 (CS1) and Table 6.6 (BCS case study).

Then, for each configuration, the following three steps are executed:

1. We compute the resource demands of the software components from the attributed FM with the mapper tool (Subsection 5.3.1).

2. We assemble the solution space CSP t_{p} with the resource demands, resource provisionings (CS1: Table 6.4; BCS case study: Table 6.7) and resource types (CS1: Table 6.1; BCS case study: Table 6.5).

3. We solve the solution space CSP with the *Z3 Theorem Prover* (Section 5.1). The result can either be a component assignment or the information that the configuration is not realisable.

The composed sets of all realisable configurations are denoted with $\mathcal{RS}^{\text{SM}}_{\text{CS1}}$ and $\mathcal{RS}^{\text{SM}}_{\text{BCS}}$. We compare those sets to the known GT ($\mathcal{RS}^{\text{GT}}_{\text{CS1}}$ and $\mathcal{RS}^{\text{GT}}_{\text{BCS}}$) with the evaluation tool (Subsection 5.3.4) and state the results according to Equation 6.1: If the compared sets are equal, e.g. they contain the *exactly same* configurations, we assume the sampling method to be evaluated positively.

$$\begin{aligned} \mathcal{RS}^{\text{SM}}_{\text{CS1}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}_{\text{CS1}} &\Leftrightarrow \text{sampling method evaluated positive for CS1,} \\ \mathcal{RS}^{\text{SM}}_{\text{BCS}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}_{\text{BCS}} &\Leftrightarrow \text{sampling method evaluated positive for BCS case study.} \end{aligned} \tag{6.1}$$

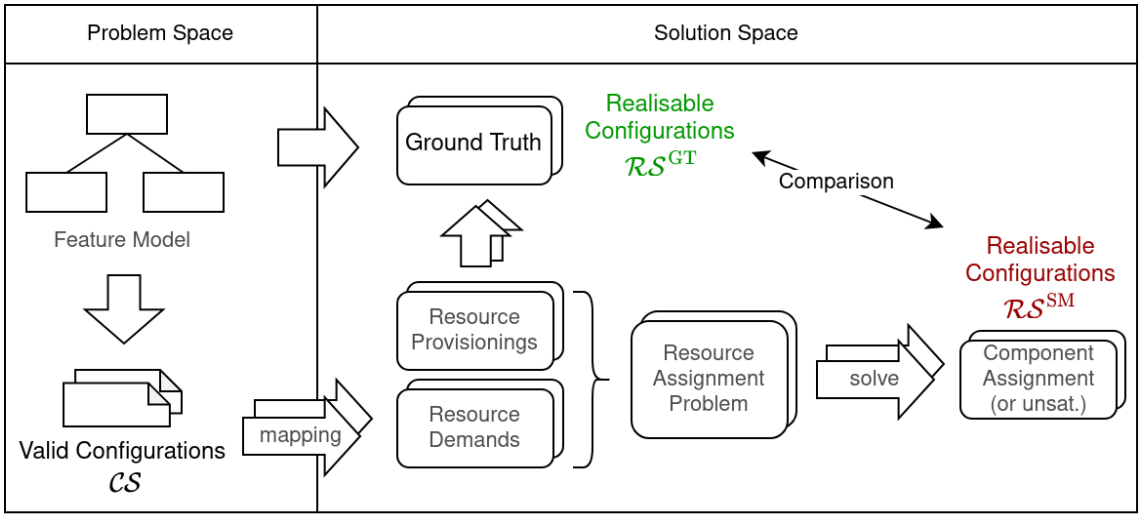

Figure 6.5: Evaluation procedure for the sampling method. (green: GT; red: computed)

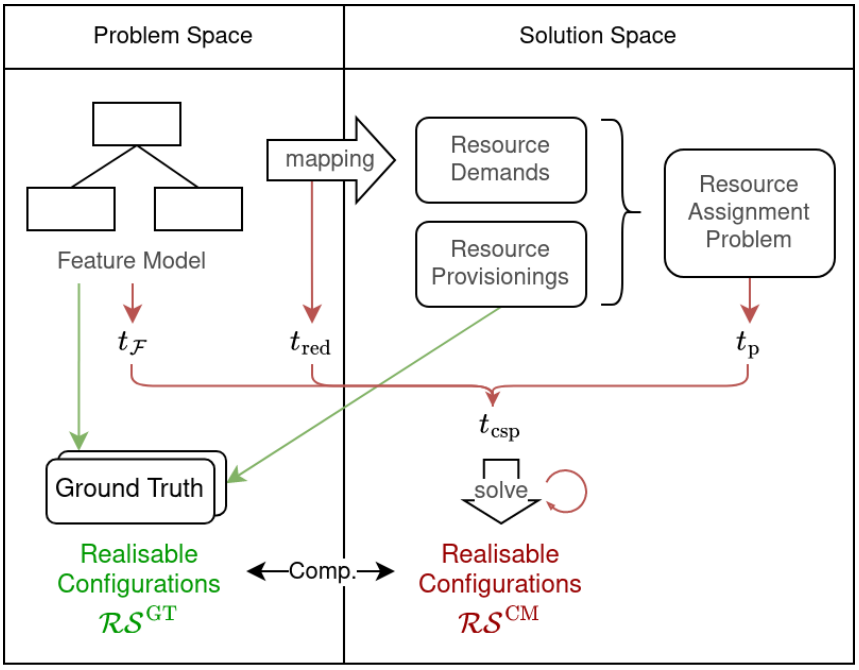

Figure 6.6: Evaluation procedure for the combined method. (green: GT; red: computed)

6.2.2 Combined Method

For the combined method, the procedure proposed in Subsection 4.3.2 is followed and shown in Figure 6.6. The problem space is formalised in the CSP $t_{\mathcal{F}}$ from the FM. The mapping and solution space are formalised in the symbolic reduce-function t_{red} and t_{p} from the resource demands and resource types. Then, these CSPs are conjuncted in the CSP t_{csp} ($t_{\text{csp}} = t_{\mathcal{F}} \wedge t_{\text{red}} \wedge t_{\text{p}}$).

We iteratively solve the CSP t_{csp} with the Z3 Theorem Prover, yielding a configuration which is both valid and realisable in each iteration. The configuration is added to the CSP t_{csp} as exclusion, so that no duplications occur in the result. As we want to compute *all* configurations both valid and realisable, the loop breaks when no more solutions are possible (t_{csp} is no longer satisfiable).

The computed set of configurations is denoted with $\mathcal{RS}^{\text{CM}}_{\text{CS1}}$ and $\mathcal{RS}^{\text{CM}}_{\text{BCS}}$ respectively. We compare those sets to the known GT with the evaluation tool (Subsection 5.3.4) and state the results according to Equation 6.2: If the compared sets are equal, e.g. they contain the *exactly same* configurations, we assume the combined method to be evaluated positively.

$$\begin{aligned} \mathcal{RS}^{\text{CM}}_{\text{CS1}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}_{\text{CS1}} \;&\Leftrightarrow\; \text{combined method evaluated positive for CS1,} \\ \mathcal{RS}^{\text{CM}}_{\text{BCS}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}_{\text{BCS}} \;&\Leftrightarrow\; \text{combined method evaluated positive for BCS case study.} \end{aligned} \tag{6.2}$$

6.3 Results & Limitations

The evaluation of the concepts we introduced in Chapter 4 was done by two case studies, CS1 and the BCS case study. The first case study serves as a verification of the proposed logical formulas and the second to evaluate the concepts on a larger, already existing PL example. We extended the model data of the case studies in Section 6.1 to fit as an initial point for the evaluation.

As this thesis focuses on the consistency of the problem and solution space of a PL, the evaluation mainly addresses the two proposed methods to decide this consistency: The sampling method (Subsection 4.3.1) and the combined method (Subsection 4.3.2). Section 6.2 presents our procedures to evaluate these methods along the case studies. The computed sets of configurations both valid and realisable are compared to the known GT in terms of set equality. In Equation 6.1 and Equation 6.2, we gave a mathematical criteria to decide whether we evaluate the methods positively or negatively.

6.3.1 Results

Table 6.8 shows the evaluation results for both case studies. Initially, the number of only-valid configurations is $|\mathcal{CS}_{\text{CS1}}| = 20$ for CS1 and $|\mathcal{CS}_{\text{BCS}}| = 7512$ for the BCS case study. The derived GT expected the number of configurations both valid and realisable to be $|\mathcal{RS}^{\text{GT}}_{\text{CS1}}| = 4$ for CS1 and $|\mathcal{RS}^{\text{GT}}_{\text{BCS}}| = 4200$ for the BCS case study.

Case Study		**CS1**	**BCS**
Valid Configurations ($\lvert\mathcal{CS}\rvert$)		20	7512
Realisable Configurations	Ground Truth ($\lvert\mathcal{RS}^{\text{GT}}\rvert$)	4	4200
	Sampling Method ($\lvert\mathcal{RS}^{\text{SM}}\rvert$)	4	4200
	Equality ($\mathcal{RS}^{\text{SM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}$)	**True**	**True**
	Combined Method ($\lvert\mathcal{RS}^{\text{CM}}\rvert$)	4	4200
	Equality ($\mathcal{RS}^{\text{CM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}$)	**True**	**True**

Table 6.8: Quantitative results of the sampling method and the combined method compared to the GT for both case studies.

With the sampling method, we computed $|\mathcal{RS}^{\text{SM}}_{\text{CS1}}| = 4$ configurations for CS1 and $|\mathcal{RS}^{\text{SM}}_{\text{BCS}}| = 4200$ configurations for the BCS case study. With the comparison of those sets to the GT with the $\equiv_{\mathcal{CS}}$-operator, we found an exact coverage of the expected results ($\mathcal{RS}^{\text{SM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}$) for both case studies. This means that the sampling method computed the *exactly same* set of configurations as we derived in the GT. We therefore assume the sampling method to be evaluated positively.

With the combined method, the number of configurations which are both valid realisable was computed to $|\mathcal{RS}^{\text{CM}}_{\text{CS1}}| = 4$ configurations for CS1 and $|\mathcal{RS}^{\text{CM}}_{\text{BCS}}| = 4200$ configurations for the BCS case study. For both sets, we found an exact coverage of the GT ($\mathcal{RS}^{\text{CM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\text{GT}}$) and therefore assume the combined method to be evaluated positively.

For both case studies, the computed sets of the sampling method and the combined method are both equal to the expected set of the GT ($(\mathcal{RS}^{\mathrm{SM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\mathrm{GT}}) \wedge (\mathcal{RS}^{\mathrm{CM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\mathrm{GT}})$). Hence, we can imply the equality of both computed sets ($\Rightarrow \mathcal{RS}^{\mathrm{SM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\mathrm{CM}}$) per case study. This means that, under the given evaluation setup, both methods yield the exactly same results.

6.3.2 Limitations

The evaluation has limitations on the statements, which are implied from the results. We discuss them in the following and give concise restrictions on these statements.

Integer Values

From a technical perspective, the contributions are assessed only with positive integer values (domain $\mathbb{Z}$) introduced with the case studies in Section 6.1. We defined the conceptual ideas in Chapter 4 generally for real values (domain $\mathbb{R}$) and therefore assume them to work also for negative, non-integer values. The tool to solve CSPs then must support real-value arithmetics. Numerical limitations should be kept in mind for very large or very small numbers, which approach technical borders (e.g. $\geq 2^{32}$ or $\geq 2^{64}$).

Application to Large Product Lines

The second technical limitation is the application of the concepts to a *large* PL. The number of valid configurations can increase exponentially to the number of features in the FM. The decision about the consistency of the problem and solution space can therefore be infeasible due to computational complexity.

The sampling method, for instance, consists of the sampling of all valid configurations and the decision about their realisability. We assume a naive sampling of all valid configurations to be in space complexity $\mathcal{O}(2^{\mathrm{poly}(|\mathcal{F}|)})$ (EXPSPACE) and the computation of the solution space CSP t_{p} to be in time complexity $\mathcal{O}(\mathrm{poly}(|\mathcal{SW}| * |\mathcal{HW}| * |\mathcal{R}|^2))$ (PTIME) per configuration [KI98]. The evaluation assesses the feasibility of the computations only for the reasonable sized PLs of the case studies, where we could compute the results without limitations.

Coverage of Problem & Solution Space

In the design of the evaluation, we stated two premises: First, the sampling method is evaluated with full coverage of the set of all possible only-valid configurations, not a subset. Second, the combined method is evaluated with full coverage of the set of configurations both valid and realisable, without restrictions on computation-time or -space. The implication on the set equality $\mathcal{RS}^{\mathrm{SM}} \equiv_{\mathcal{CS}} \mathcal{RS}^{\mathrm{CM}}$ is only given, if both premises are fulfilled. Otherwise, no statement on this set equality can be made, as the combined method then generates an arbitrary, non-influenceable result set.

Evaluation of Resource & Component Assignments

The evaluation decides the equality of the result sets $\mathcal{RS}^{\mathrm{SM}}$ and $\mathcal{RS}^{\mathrm{CM}}$ to the GT $\mathcal{RS}^{\mathrm{GT}}$. It does not assess the actual assignments of resource demands from software components to resource provisionings of hardware components explicitly. Implicitly, a valid resource assignment implies a valid component assignment, which itself implies the given realisability of the configuration. As the contradiction to these two implications also holds, we assume that our decision about the set equality suffices to state the positive evaluation for the contributions.

6.4 Discussion & Relation to Research Questions

In Section 6.3, we presented the results of the evaluation and its implications. The evaluation is focused on the two proposed methods to decide the consistency of the problem and solution space: The sampling method (Subsection 4.3.1) and the combined method Subsection 4.3.2. In the following, we relate the contributions and their evaluation to the research questions stated in Chapter 1.

Research Question RQ1

Research question RQ1 addresses the description of solution space artefacts in a PL. We contributed a meta-model as an extension to the UCM from Wittler et al. [WKR22], which relies on components and resources to specify demands and provisionings. In the evaluation, we used two case studies, which are instances of our meta-model and thus represent the meta-model in the data. Because we modeled the case studies along the proposed meta-model (and not vice-versa), we can not state implications on the meta-model itself after the evaluation of the case studies, as this would create a circular closure.

Research Question RQ2

Research question RQ2 addresses the decision about the realisability of a valid configuration. In Section 4.2, we proposed a formalisation of solution space artefacts as a CSP. By solving this CSP, we can either imply the realisability of the configuration with a valid component assignment, or the non-realisability because the CSP is unsatisfiable. The sampling method and the combined method rely on this formalisation to compute a set of configurations which are both valid and realisable.

In the evaluation, we derived the correctness of these two methods by applying the quantitative data of the two case studies. Every valid configuration was decided correctly about its realisability by both methods. Therefor, we assume the contributions to research question RQ2 to be evaluated positively under the given evaluation setup and limitations discussed in Section 6.3.

Research Question RQ3

Research question RQ3 addresses the description of run-time variability in a PL. In Section 4.4, we proposed a modification to our meta-model which introduces capabilities. Capabilities can express dependencies between solution space artefacts. We presented two demonstrations how to shift these dependencies into the problem space and express them with concepts already known.

The first demonstration covers the dependency between a software and a hardware component. We showed how this dependency can be represented in the problem space by designing a resource type and according resource demands and provisionings. Then, it is part of the procedure we proposed in Section 4.2 to decide the realisability of a configuration. Hence, the evaluation of dependencies between software and hardware components is covered by the evaluation of research questions RQ1 and RQ2, because it only relies on these contributions.

The second demonstration covers the dependency between two software components. We showed how this dependency can be represented completely by FM constraints

in the problem space, independent from the solution space. Therefore, dependencies between software components can be analysed by known problem space analysis methods [BSRC10].

We consider the other cases, the dependency from a hardware to a software component and the dependency between two hardware components, as not meaningfully applicable to our meta-model. They either contradict to the pattern of designing demands and dependencies along user-experienceable behaviour (features) in the problem space of a PL or are a task of plain hardware engineering. Therefore, we assume all meaningful cases to be reduced to already evaluated contributions of this thesis.

Main Contribution

The main contribution of this thesis is the decision of the consistency between the problem and solution space of a PL. With the sampling method and the combined method, we proposed two procedures to compute the set of all configurations both valid and realisable. We designed them to cover this set of configurations *completely* (full coverage): The sampling method uses *all* valid configurations (not a subset) to decide about the realisability of each configuration. The combined method directly computes the complete set without further breaking conditions, such as execution time or result set size.

In the evaluation, we derived the GT of both case studies the way that this full coverage was expected of both methods. The results of the evaluation show that both methods satisfy the expectations, yielding exactly the set of configurations expected. Therefore, we assume this main contribution of the thesis as given in theory under the (possibly) practical limitations due to computational complexity.

6.5 Threats to Validity

In the following, we discuss threats to the validity of the evaluation. Internal threats concern the setup and execution of the evaluation and are discussed in Subsection 6.5.1. External threats concern the generalisability of the statements and are discussed in Subsection 6.5.2.

6.5.1 Internal Threats

Evaluation of Resource & Component Assignments

The evaluation focuses on the comparison of the computed sets of configurations ($\mathcal{RS}^{\text{SM}}_{\text{CS1}}$, $\mathcal{RS}^{\text{CM}}_{\text{CS1}}$, $\mathcal{RS}^{\text{SM}}_{\text{BCS}}$, $\mathcal{RS}^{\text{CM}}_{\text{BCS}}$) to the expected sets of configurations ($\mathcal{RS}^{\text{GT}}_{\text{CS1}}$, $\mathcal{RS}^{\text{GT}}_{\text{BCS}}$). Thus, the computed decision about the realisability of each configuration is compared to the expected decision. The defined solution space CSP (Section 4.2) used in the sampling method and in the combined method actually computes assignments of demanded resources from software components to provided resources of hardware components. These resource assignments are not explicitly evaluated. Furthermore, the component assignments, which are implied by the resource assignments, are also not explicitly evaluated. Nevertheless, in Section 6.3 we described how correct resource assignments imply a correct decision about the realisability of a configuration. Therefore we argue that an incorrect resource assignment would yield an incorrect decision about the realisability of a configuration, which would be detectable with our evaluation design.

Correctness of Implemented Code Artefacts

For supporting the evaluation procedure, we implemented tools described in Section 5.3: An implementation of the reduce-function, implementations of the sampling method and the combined method as well as an implementation of the $\equiv_{\mathcal{CS}}$-operator. These tools could contain errors which could lead to wrong computation outputs. As the practical results of the evaluation are equal to the theoretical expectations, we assume our tools to be free of errors.

6.5.2 External Threats

Lack of Formal Proofs

The contributions proposed in Chapter 4 consist of a number of propositional logical formulas, which form CSPs of the solution space (for the sampling method) and the combined method. With the application on case studies and the comparison against a GT, we showed the correctness of the results of these formulas, but not their *formal* correctness. Hence, we can only state a positive evaluation of the contributions in context of the case studies.

Lack of External Validation

Furthermore, we designed the case studies without references to existing work. In detail, the BCS case study relies on an existing FM but was extended by solution space artefacts. As our contribution and evaluation focuses on these solution space artefacts, a circular closure is built, where the expected results are designed in advance. To *validate* the contributions, they should be applied to an external example of an instantiated meta-model.

Technical Threats for Generalisability of Contributions

In Section 6.3, we discussed two technical limitations of our contributions: First, they were applied only to positive integer values (domain $\mathbb{Z}$). We argue that our contributions can be generalised to a superset of $\mathbb{Z}$, because we already defined them for real values (domain $\mathbb{R}$) in Chapter 4. Second, the application of the concepts to large PLs could be infeasible due to exponential complexity of the computations. We suggest to adapt the sampling method to be applied to a representative, real subset of all valid configurations ($\mathcal{CS}_{\text{sampled}} \subsetneq \mathcal{CS}$) while maintaining a coverage criteria. We defined the formal conditions of this adaption in Table 4.4.

7. Related Work

This chapter presents contributions from other authors which are related to this thesis. We searched literature in the combined contexts of *Variability* (e.g. PLs and other product artefacts with reuse) and *Implementation* (e.g. implementation-, solution space- or testing artefacts) and identified ten papers of interest. Most of the work (6 of 10) we reviewed focuses on (source) code as implementation artefacts. We categorized all papers (non-disjunctively) to four topics: Models of solution space artefacts (Section 7.1, 3 papers), analysis techniques of solution space artefacts (Section 7.2, 3 papers), papers concerning the mismatch between problem and solution space (Section 7.3, 3 papers) and sampling strategies (Section 7.4, 4 papers).

7.1 Modeling of Solution Space Artefacts

Ananieva et al. [AGK+22] proposed the UCM as a meta-model of a PL, representing variability in space (variants) and time (versions). A solution space model to the UCM was proposed from Wittler et al. [WKR22] and serves as a foundation of this thesis. It introduced the concepts of components and resources. Our contributions refined and extended this meta-model in three aspects: First, we modified resources the way that their properties (additivity, exclusivity, boundary) are only defined by a resource type, serving as an interface to the characteristics and behaviour of resources. Second, we proposed a formalisation of solution space artefacts as CSP, allowing for automatic reasoning about the realisation of configurations. Third, we detailed capabilities as a representation of run-time variability in the solution space of a PL.

Neema et al. [NSKB03] suggested a meta-model and tool to synthesize products in the domain of embedded systems engineering. It abstracts over domain specific modeling languages (e.g. the UML) by providing a framework which can deal with arbitrary modeling languages. Therefore, a user instantiates a modeling language (e.g. a UML class diagram) and defines the model, syntax and semantics of the instance. The proposed tool supports in finding products that meet all constraints of the defined model and semantics. In comparison to this thesis, Neema et al.

[NSKB03] contributed a generic interface to constraint satisfaction for variability-aware products, while this thesis contributes an actual meta-model and constraint satisfaction method.

7.2 Analysis of Solution Space Artefacts

The reviewed papers about the analysis of solution space artefacts mostly focus on checking incompatibilities in a conditional set of implementation artefacts, depending on the input configuration. Hentze et al. [HSTS22] integrated solution space constraints (e.g. dependencies between artefacts) into the problem space FM (the *Integrated Feature Model*) by handling the solution space artefacts as features of distinguishable type.

The work of Kästner et al. [KGR+11] and Gazillo and Grimm [GG12] contribute analysis tools for implementation artefacts (source code) of variability-aware software systems which do not have to generate each software product individually, but can detect errors in advance of preprocessor actions.

While Kästner et al. [KGR+11] and Gazillo and Grimm [GG12] focus their contributions on the analysis of source code, the Integrated Feature Model from Hentze et al. [HSTS22] allows the application to arbitrary domains. Nevertheless, all three contributions are similar to this thesis in terms of relating explicit solution space constraints to the variability of a product family. They differ to this thesis either in their generality of application (Kästner et al. [KGR+11], Gazillo and Grimm [GG12]) or their ability to express more complex solution space constraints than basic propositional logic as used in FMs (Hentze et al. [HSTS22]).

7.3 Inconsistencies Between Problem and Solution Space

Contributions about the inconsistency of problem and solution space regard the explicit distinction between variability along the FM and variability in derived products. They share the initial assumption that the solution space contains further constraints which diminish the variability of a PL.

As described in the previous section, Hentze et al. [HSTS22] integrated solution space constraints as (solution-space-) features into the problem space of a PL. Their work further allows for a comparison of the variability with and without these solution-space-features, quantifying a possible inconsistency between problem and solution space.

The contributions of Thüm et al. [TKES11] and El-Sharkawy et al. [ESKS17] reason about inconsistencies caused by feature model design. The argumentation of Thüm et al. [TKES11] is based on abstract features, which do not have any solution space artefacts and only exist for structural reasons in a FM. They argue that, due to these abstract features, the number of actual, distinctive product variants is lower than the number of valid configurations. El-Sharkawy et al. [ESKS17] analysed the Linux kernel for mismatches between its variability model (problem space) and its actual variability. Their results show, that most of the mismatches rely on variability options which do not effect the resulting product. This corresponds highly to the work of Thüm et al. [TKES11].

Thüm et al. [TKES11] and El-Sharkawy et al. [ESKS17] related inconsistencies to the design of the FM and the derived solution space artefacts and constraints. In contrast, Hentze et al. [HSTS22] let solution space constraints to be stated independently from the problem space, but integrated them later into the FM. All three contributions are able to quantify inconsistencies between problem and solution space. With the sampling method, our contributions are also able to quantify this mismatch. It differs from the related work, because it keeps problem and solution space constraints completely separate.

7.4 Configuration and Product Sampling

Configuration sampling is introduced in Subsection 3.1.2. It addresses the generation of a representative subset of configurations, called *Sample*. A basic sampling strategy is the *T-wise Feature Interaction Coverage*, where all possible, valid t-wise combinations of features are regarded. The enhanced implementation of the naive procedure is called *YASA* from Krieter et al. [KTS+20] and is, for example, available in the *FeatureIDE*. Varshosaz et al. [VAHT+18] gave a general overview of sampling strategies in the literature. We want to focus on contributions which also regard solution space artefacts in their sampling strategy.

A sampling strategy for generic solution space artefacts was proposed by Hentze et al [HPS+22] and relies on their previous work on the Integrated Feature Model. Their sampling strategy takes solution space constraints and behaviour into account when sampling for t-wise feature interaction coverage. For example, two t-wise feature interactions can yield the same solution space artefacts and thus differ only in the problem space, not in the solution space where further testing is relevant. Therefore, the sample set size can be reduced by avoiding the duplication of configurations with the same set of solution space artefacts.

A similar approach for the application on PLs with source code artefacts is taken by Kim et al. [KBK11] and Shi et al. [SCD12]. Both contributions analyse, if different configurations yield differences in the resulting source code of the derived software products. Kim et al. [KBK11] therefore use static software analysis tools on each derived product, while Shi et al. [SCD12] reason with feature dependencies and interactions resulting from code analysis in advance.

Tartler et al. [TLD+11] proposed a sampling strategy that regards source code coverage, e.g. how much lines of code a sample covers. First, they build a graph structure from source code blocks (nested blocks and if-else-cascades). Then, they identify source code blocks which can not be selected the same configuration. These are the solution space constraints. With graph theory algorithms (coloring problem solver), they find combinations of source code blocks representing configurations, and therefore achieve a high statement coverage.

All four contributions (Hentze et al. [HPS+22], Kim et al. [KBK11], Shi et al. [SCD12], Tartler et al. [TLD+11]) share the intention to reduce the sample set size by analysing similarities in sets of solution space artefacts, as different configurations in the problem space do not necessarily yield different product variants. With the sampling method, our contributions can also reduce a sample set size. The main difference to the related work is, that the sampling method identifies *incompatibilities* between solution space artefacts instead of *similarities* between product variants.

8. Conclusion and Outlook

This thesis addresses the consistency between the problem and solution space of a PL in CPSs. The problem space diverges from the solution space, if the variability of the FM (problem space) is reduced by solution space constraints, such as incompatible solution space artefacts. We proposed two methods to decide this consistency, the sampling method and the combined method. The sampling method (Subsection 4.3.1) takes a sample of valid configurations and decides about the realisability for each configuration. This allows to check in advance, if an existing configuration is still working after an update. The combined method (Subsection 4.3.2) generates configurations which are both valid and realisable, directly. This has practical advantages for further product testing, as usually only these configurations are of interest.

In detail, the thesis was structured along three research questions introduced in Chapter 1:

RQ1 How can solution space artifacts be formally described in a meta-model?

RQ2 How can the realisability of a valid configuration be decided?

RQ3 How can run-time variability be described?

For research question RQ1, we modified the UCM from Wittler et al. [WKR22] to express generic solution space artefacts (Section 4.1). It consists of software components which demand resources and hardware components which provide resources. A resource belongs to a resource type, which has three properties: Additivity, exclusivity and a boundary type. These properties control the behaviour of the resource, for example if it adds up in value when demanded multiple times, such as RAM.

For research question RQ2, we proposed a procedure to formalise the solution space artefacts of a valid configuration as a CSP (Section 4.2). It is based on the modified UCM and represents a resource allocation problem, where the resource demands of the software components are compared to the resource provisionings of the hardware

components. If all resource demands are satisfied by the resource provisionings, we consider the configuration to be realizable.

For research question RQ3, we introduced the concept of capabilities to the modified UCM (Section 4.4). Capabilities can express arbitrary dependencies between solution space artefacts, for example two software components. We showed two demonstrations how to shift these dependencies into the problem space of a PL, where they can be represented as resource demands or as FM constraints. This allows for the application of our contributions on the research questions RQ1 and RQ2.

In the evaluation (Chapter 6), we designed two case studies: The CS1 to verify the contributions and the BCS case study to apply the contributions to a well known PL in the research context. For each case study, we computed the set of configurations which are both valid and realisable with the sampling method and the combined method. Then we compared the result sets to the GT (expected sets) for equality. We found a full coverage of the expected results for both methods (sampling method and combined method) and both case studies. Therefore, we assume the two methods to be evaluated positively with the given limitations.

As a further work on this topic, we first suggest to generalise the properties of a resource type in the meta-model. With the three properties, additivity, exclusivity and boundary type, we initially enabled to control the behaviour of resources. For example, the demanded RAM of a number of software components is added up when comparing it to the provided resource of a hardware component. In real-world scenarios, resource allocation can behave non-linearly (e.g. logarithmically) due to technical limitations, such as computation overhead or latency. Hence, we suggest to generalise resource properties to the specification of arbitrary functions, instead of simple summation-arithmetic.

With the combined method, configurations both valid and realisable can be iteratively computed. Depending on the used solver (in this thesis: Z3 Theorem Prover), the resulting configuration (per iteration) can not be influenced and mainly depends on the algorithm inside the solver. Therefore, we suggest to enhance the procedure to influence the iterative computation of configurations. For the Z3 Theorem Prover, Bjorner et al. [BPF15] proposed the specification of objective functions when applying the solver to CSPs. This way, manufacturers of PLs can compute configurations which not only satisfy problem and solution space constraints but also optimize on self-defined metrics.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[AGK+22] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf Reussner. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering*, 27(5):101, September 2022.

[BH02] Endre Boros and Peter L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, November 2002.

[Bjø12] Nikolaj Bjørner. Taking Satisfiability to the Next Level with Z3. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 1–8, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BMNW21] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. Programming Z3, January 2021.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035, pages 194–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In Rudibert King, editor, *Active Flow and Combustion Control 2018*, volume 141, pages 491–503. Springer International Publishing, Cham, 2005. Series Title: Notes on Numerical Fluid Mechanics and Multidisciplinary Design.

[BZ18] Martin Becker and Bo Zhang. How Do Our Neighbours Do Product Line Engineering? A Comparison of Hardware and Software Product Line Engineering Approaches from an Industrial Perspective. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, pages 190–195, New York, NY, USA,

2018. Association for Computing Machinery. event-place: Gothenburg, Sweden.

[CN02] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. The SEI series in software engineering. Addison-Wesley, Boston, 2002.

[CRSC09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of satisfiability*, number v. 185 in Frontiers in artificial intelligence and applications, pages 825–885. IOS Press, Amsterdam, The Netherlands ; Washington, DC, 2009. OCLC: ocn290492523.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM*, 54(9):69–77, September 2011. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[ESKS17] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. An Empirical Study of Configuration Mismatches in Linux. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, pages 19–28, Sevilla Spain, September 2017. ACM.

[GG12] Paul Gazzillo and Robert Grimm. SuperC: parsing all of C by taming the preprocessor. *ACM SIGPLAN Notices*, 47(6):323–334, August 2012.

[GZ17] Weiwei Gong and Xu Zhou. A survey of SAT solver. *AIP Conference Proceedings*, 1836(1):020059, June 2017.

[HPS+22] Marc Hentze, Tobias Pett, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Ina Schaefer. 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, pages 135–147, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Auckland, New Zealand.

[HSTS22] Marc Hentze, Chico Sundermann, Thomas Thüm, and Ina Schaefer. 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, pages 322–333, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Montreal, Quebec, Canada.

[JTDL98] Jianxin Jiao, Mitchell M. Tseng, Vincent G. Duffy, and Fuhua Lin. Product family modeling for mass customization. *Computers & Industrial Engineering*, 35(3-4):495–498, December 1998.

[KBK11] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, pages 57–68, New York, NY, USA, 2011. Association for Computing Machinery. event-place: Porto de Galinhas, Brazil.

[KGR+11] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 805–824, Portland Oregon USA, October 2011. ACM.

[KI98] Naoki Katoh and Toshihide Ibaraki. Resource Allocation Problems. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization: Volume1–3*, pages 905–1006. Springer US, Boston, MA, 1998.

[KTS+20] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. YASA: yet another sampling algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–10, Magdeburg Germany, February 2020. ACM.

[LAMS05] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool Support for Feature-Oriented Software Development: FeatureIDE: An Eclipse-Based Approach. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '05, pages 55–59, New York, NY, USA, 2005. Association for Computing Machinery. event-place: San Diego, California.

[Law01] Eugene L. Lawler. Bipartite Matching. In *Combinatorial optimization: networks and matroids*, pages 186–187. Dover Publications, Mineola, N.Y, 2001.

[Mil72] Raymond E. Miller. Reducibility among Combinatorial Problems. In Richard M. Karp, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.

[Nat] National Science Foundation. Cyber-Physical Systems (CPS).

[Nor08] Linda Northrop. Software Product Lines Essentials, 2008.

[NSKB03] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-Based Design-Space Exploration and Model Synthesis. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Rajeev Alur, and Insup Lee, editors, *Embedded Software*, volume 2855, pages 290–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Series Title: Lecture Notes in Computer Science.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[RND10] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[SCD12] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Juan De Lara, and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212, pages 270–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[SDS04] Z. Stojanovic, A. Dahanayake, and H. Sol. Modeling and design of service-oriented architecture. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, volume 5, pages 4147–4152, The Hague, Netherlands, 2004. IEEE.

[SRMI12] Sasha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study. Technical Report 2012-07, Technische Universität Braunschweig, 2012.

[SSG+22] Marc Schindewolf, Hannes Stoll, Houssem Guissouma, Andreas Puder, Eric Sax, Andreas Vetter, Marcel Rumez, and Jacqueline Henle. A Comparison of Architecture Paradigms for Dynamic Reconfigurable Automotive Networks. In *2022 International Conference on Connected Vehicle and Expo (ICCVE)*, pages 1–7, Lakeland, FL, USA, March 2022. IEEE.

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):1–45, 2014. Publisher: ACM New York, NY, USA.

[TKES11] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *2011 15th International Software Product Line Conference*, pages 191–200, Munich, Germany, August 2011. IEEE.

[TLD+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, pages 1–5, Cascais Portugal, October 2011. ACM.

[VAHT+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, pages 1–13, New York, NY, USA, 2018. Association for Computing Machinery. event-place: Gothenburg, Sweden.

[WKR22] Jan Willem Wittler, Thomas Kühn, and Ralf Reussner. 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*, pages 94–98, Graz Austria, September 2022. ACM.

A. Appendix

A.1 Case Study 1 (CS1) Data

Next page.

Feature	Software Component (sw_i)	Demand (rd^i_k)
`infotainment-small`	`infotainment-software` (sw_0)	$rd^0_0 = 5$ $rd^0_1 = 100$ $rd^0_2 = 1$ $rd^0_6 = 2$ $rd^0_7 = 1$
`infotainment-big`	`infotainment-software` (sw_0)	$rd^0_0 = 10$ $rd^0_1 = 100$ $rd^0_2 = 3$ $rd^0_6 = 6$ $rd^0_7 = 3$
`critical-component-monitoring`	`ccm-visualisation-software` (sw_1)	$rd^1_4 = 1$ $rd^1_6 = 2$ $rd^1_8 = 1$
	`ccm-calculation-software` (sw_2)	$rd^2_3 = 2$ $rd^2_6 = 1$
`electric-seats`	`seat-adjustment-panel-software` (sw_3)	$rd^3_5 = 1$
`seat-adjustment-6way`	`seat-adjustment-panel-software` (sw_3)	$rd^3_9 = 6$
`seat-adjustment-10way`	`seat-adjustment-panel-software` (sw_3)	$rd^3_9 = 10$
`seat-motorisation-front`	`seat-adjustment-panel-software` (sw_3)	$rd^3_{10} = 2$
`seat-motorisation-front-back`	`seat-adjustment-panel-software` (sw_3)	$rd^3_{10} = 4$
`tyre-pressure-monitoring`	`tyre-pressure-monitoring-software` (sw_4)	$rd^4_{11} = 4$

Table A.1: Resource demands of features in CS1.

A.2 Comfort System (BCS) Case Study Data

Next page.

Feature	Software Component (sw_i)	Demands (rd_k^i)
led-alarm-system	hmi-software (sw_0)	$rd_0^0 = 2$
led-finger-protection	hmi-software (sw_0)	$rd_0^0 = 1$
led-central-locking	hmi-software (sw_0)	$rd_0^0 = 1$
led-power-window	hmi-software (sw_0)	$rd_0^0 = 1$
led-exterior-mirror	hmi-software (sw_0)	$rd_0^0 = 1$
led-em-heatable	hmi-software (sw_0)	$rd_0^0 = 1$
em-electric	exterior-mirror-control-software (sw_1)	$rd_7^1 = 10$ $rd_1^1 = 5$
em-heatable	exterior-mirror-control-software (sw_1)	$rd_1^1 = 20$
pw-finger-protection	power-window-control-software (sw_2)	$rd_6^2 = 1$
pw-manual	power-window-control-software (sw_2)	$rd_6^2 = 2$
pw-automatic	power-window-control-software (sw_2)	$rd_7^2 = 5$ $rd_6^2 = 3$
rc-key	security-software (sw_3)	$rd_2^3 = 2$ $rd_3^3 = 10$
rck-pw-automatic	power-window-control-software (sw_2)	$rd_7^2 = 5$
	security-software (sw_3)	$rd_2^3 = 2$
rck-exterior-mirror	exterior-mirror-control-software (sw_1)	$rd_7^1 = 10$
	security-software (sw_3)	$rd_2^3 = 2$
rck-alarm-system	security-software (sw_3)	$rd_2^3 = 2$
rck-safety-function	security-software (sw_3)	$rd_2^3 = 2$
alarm-system	security-software (sw_3)	$rd_3^3 = 100$ $rd_5^3 = 4$
alarm-system-interior	security-software (sw_3)	$rd_3^3 = 700$ $rd_5^3 = 1$
central-locking	security-software (sw_3)	$rd_3^3 = 10$
central-locking-automatic	security-software (sw_3)	$rd_4^3 = 1$

Table A.2: Resource demands of features in the BCS case study.