Karlsruhe Institute of Technology

Test, Validation, Analysis

Implementing a System Generation Aware Unified Conceptual Model

Bachelor's Thesis
of

Lennart Rak

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. Reussner
Advisors:	M.Sc. Tobias Pett
	M.Sc. Philip Ochs

Completion period: 26. January 2024 – 17. May 2024

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 16. May 2024

Zusammenfassung

Kunden erwarten heutzutage hochkonfigurierbare Produkte, die mit Hilfe von Produktlinien entwickelt werden können. Produktlinien beschreiben verschiedene Produktvarianten und Versionen dieser Varianten und drücken damit die Variabilität im Raum durch Varianten und die Variabilität in der Zeit durch Versionen aus. Eine Produktlinie kann in einen Problemraum, der die für die Nutzer sichtbaren Eigenschaften oder Verhaltensweisen eines Produktes, die sogenannten Features, und deren Abhängigkeiten beschreibt, und einen Lösungsraum, der die Realisierungsartefakte darstellt, unterteilt werden. Der Stand der Technik lässt folgende ungelöste Herausforderungen erkennen: das Fehlen einer Validierung und Prüfung der Realisierbarkeit von Varianten im Zeitverlauf sowie das Fehlen eines einzigen Instruments, das eine Lösung für diese Herausforderung bietet und zudem mit etablierten Werkzeugen kompatibel ist. In dieser Arbeit gehen wir die beschriebenen Herausforderungen an, indem wir ein bestehendes Metamodell, das Unified Conceptual Model (UCM), erweitern. Dafür bieten wir eine Implementierung des UCM an, die wir durch die Integration bestehender Konzepte zum Ausdruck von Variabilität im Raum erweitern, um unsere Implementierung an bekannte Werkzeuge, wie FeatureIDE, anzupassen. Wir integrieren auch die Konzepte der Deltamodellierung, um den Unterschied zwischen zwei Versionen im UCM zu beschreiben, und wir verwenden das Modell der System Generation Engineering (SGE)-Variationsarten, um Änderungen für die weitere Analyse mit Nominaltypen zu kategorisieren. Wir zeigen die Anwendbarkeit unserer Implementierung, indem wir eine Realisierungsanalyse an einer Erweiterung der bekannten Body Comfort System (BCS) Case Study durchführen. Wir verwenden das erweiterte BCS, um zu zeigen, dass wir die SGE-Variationsarten auf ein domänenübergreifendes Produktline anwenden können, und zeigen, dass unsere UCM-Implementierung den semantischen Unterschied zwischen zwei Versionen korrekt beschreibt, indem wir eine Realisierungsanalyse unter Verwendung der UCM-Instanz durchführen, die die Version durch Deltas beschreibt.

Abstract

Nowadays, customers expect highly configurable products, which can be developed using the approach of product lines. Product lines describe different product variants and versions of these variants, thus expressing variability in space through variants and variability in time through versions. A product line can be divided into problem space, where the user-visible characteristics or behaviours of a product, called features, and their dependencies are described, and the solution space, which represents the realisation artefacts. The state of the art reveals the following unsolved challenges: the lack of validation and realisability testing of variants over time, and the lack of a single instrument that provides a solution to this challenge and is also compatible with established tools. In this thesis, we approach the described challenges by extending an existing metatmodel, the UCM. Therefore, we propose an implementation of the UCM, which we extend by integrating existing concepts for expressing variability in space in order to adapt our implementation to known tools, such as FeatureIDE. We also integrate the concepts of delta modelling to describe the difference between two versions in the UCM, and we use the model of SGE variation types to categorise changes for further analysis with nominal types. We show the applicability of our implementation by reproducing a realisation analysis on an extension of the well-known BCS Case Study. We use the extended BCS to show that we can apply the SGE variation types to a cross-domain product line, and show that our UCM implementation correctly describes the semantic difference between two versions by performing a realisation analysis using the UCM instance that describes the version by deltas.

Contents

List of Figures

List of Tables

Acronyms

RQ	Research Question
SG	Subgoal
GT	Ground Truth
CS	Case Study
RAM	random access memory
GB	gigabyte
ECU	electronic computing unit
CNF	conjunctive normal form
XML	Extensible Markup Language
XMI	Extensible Markup Language (XML) Metadata Interchange
CSV	comma-separated values
PL	product line
SUM	Single Underlying Model
VSUM	Virtual Single Underlying Model (SUM)
UCM	Unified Conceptual Model
FM	feature model
FD	feature diagram
SGE	System Generation Engineering
SGEM	SGE Model
PV	Principle Variation
CV	Carryover Variation
EV	Embodiment Variation
BCS	Body Comfort System
EMF	Eclipse Modeling Framework

AS	Alarm System
CLS	Central Locking System
EM	Exterior Mirror
FP	Finger Protection
HMI	Human Machine Interface
PW	Power Windows
RCK	Remote Control Key

1. Introduction

Product lines (PLs) are a way of organising the development of software and hardware products that share a common set of configuration options and components [NC13]. This approach can be used to reduce development costs, time-to-market and improve product quality by utilising common components and considering variability early in the development process. Variability refers to the potential for multiple product variants and versions of variants, which is a common challenge in the development of mass-customized hardware and software products, such as automotive, robotics, embedded systems, and production systems [ABKS13].

PLs can be divided into problem space and solution space. The problem space describes the user visible characteristics or behaviours of the product, called features, and their dependencies in the feature model. The solution space represents the realisation artefacts of the features and their dependencies between those artefacts. The artefacts are related to the features in the problem space, linking the conceptual definition in the problem space to the concrete implementation in the solution space [ABKS13].

In the problem space, the features of a PL and their dependencies are described in a feature model (FM). An example for a FM of a `Car` PL is shown in Figure 1.1. The first version of the `Infotainment` subsystem of the `Car` (left side of Figure 1.1), requires the selection of the `Radio` feature, the `Navigation` feature, or both features at the same time. If the `Navigation` feature is selected the `Voice Navigation` feature can be chosen optionally. This results in five possible configurations of the `Infotainment` subsystem. The existence of different variants of one system at a certain point in time is called variability in space. The challenge of variability in space is to manage the almost infinite number of configurations that result from the combinatorial explosion as the number of features in the product line increases [KTS+20].

The transformation of a valid configuration in the problem space into a concrete product in the solution space is a process known as realization. Therefor it is necessary to consider the problem space and the solution space together, which intensifies the challenges of managing variability in space. Figure 1.1 shows that the `Radio` feature, the `Navigation` feature and the `Voice Navigation` feature are developed on

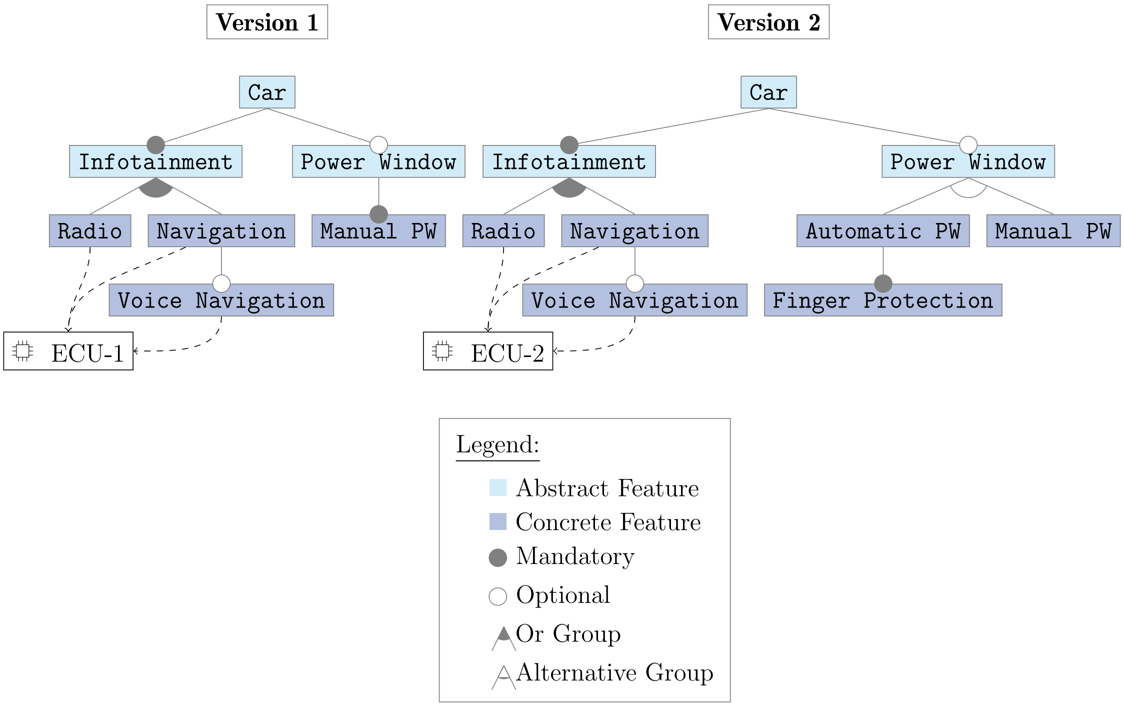


Figure 1.1: FM of a car PL

the same electronic computing unit (ECU), which may cause realisability issues of some configurations. For instance, assume that the ECU provides 2 gigabyte (GB) of random access memory (RAM), the `Radio` feature requires 1 GB, and the `Navigation` feature requires 2 GB. Then a product variant with a radio is realisable. A variant with the `Navigation` feature can be realised as well, but as the `Radio` feature and `Navigation` feature together require 3 GB of RAM, a variant with both features is not realisable. This realisability problem can only be found if problem space and solution space are analysed together.

The Unified Conceptual Model (UCM) provides a unified representation of the problem and solution space. Figure 1.2 shows the UCM with the visualisation of the problem space on the left and the visualisation of the solution space on the right side. Connections between the left and the right side, represent relationships and dependencies between features and hardware and software components. This unified view enables the modelling of dependencies between problem space and solution space and the development of further analysis methods [WKR22].

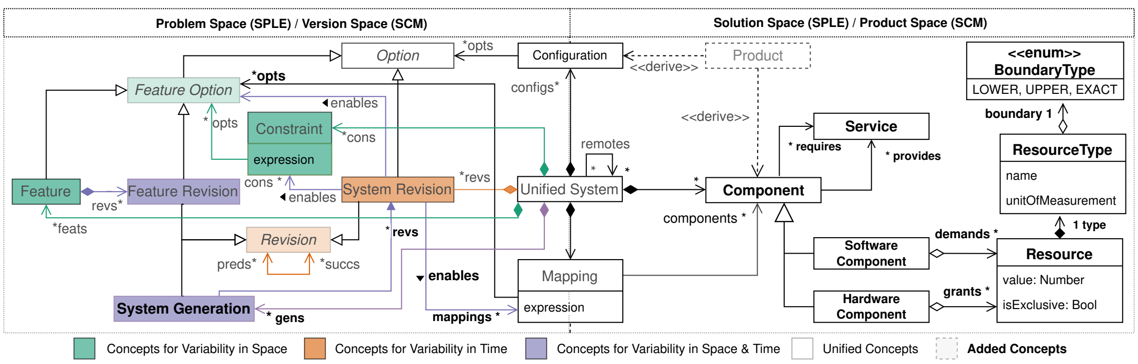


Figure 1.2: Unified Conceptual Model according to [WKR22]

Besides variability in space, product line engineering must also consider the continuous evolution of systems, called variability in time. Variability in time can lead to structural changes in the FM, affecting not only individual features but also the overall organisation and relationships between them. This dynamic evolution reflects the continuous adaptation of products to changing needs and market trends. For example, if to the mandatory `Manual PW` feature a new option, as the `Automatic PW` feature is added, the structure of the `Power Window` subsystem may need to be adjusted to consider the new dependency. This structural change can be seen in Figure 1.1 from Version 1 to Version 2.

The dimension of variability in time also brings with it the challenge of maintaining compatibility between different product versions and ensuring that they work together. The difference in time could, for example, be due to a software update that was installed to vehicles via an over-the-air update for the `Infotainment` subsystem to provide better and more advanced path finding at `Navigation` (Version 1 to Version 2 in Figure 1.1). The update requires more hardware resources, which the older versions of the `Car` PL (Version 1) cannot provide with the ECU-1. Therefore the update can only be applied to later versions of the vehicle equipped with ECU-2.

The challenges of variability are compounded by the interplay of variability in space and time, which increases the complexity of product development and maintenance. For example, a manufacturing system must support a variety of product configurations while adapting to changes in production processes and software versions.

Within the domain of mechanical engineering, the System Generation Engineering (SGE) Model (SGEM) is a descriptive model that provides a systematic approach to analyse the evolution of system generations over time. The development of new system generations in the SGEM is based on reference (sub-) system and consists of variations of (sub-) systems for reuse (Carryover Variation (CV)) or development through Embodiment Variation (EV) and Principle Variation (PV) which can change the functional principle of (sub-) systems [ABR17]. The model can be used to express and track variability in time by analyzing changes in system generations over time.

Goal of this Thesis

Analyzing the state of the art reveals the following still unsolved challenges:

- an almost infinite number of configurations
- unifying the problem space and solution space
- validation and realisability testing of variants over time
- lack of a single instrument that offers solutions for all the challenges described

In this bachelor thesis we address these four challenges by implementing tool support for an integrated management of variability in time and space. Therefore, we define the following four Subgoals (SGs), which represent the tasks to be done in this bachelor thesis.

Subgoal 1

Implementation of the UCM in the Eclipse Modeling Framework (EMF).

An instantiable model of the UCM in EMF opens up the possibility of combined analysis across problem and solution space, in particular for realisability testing as proposed by Ochs [Och23]. Our contribution also includes implementing adapters to import data from FeatureIDE and exchange formats such as CSV and exports them back to FeautreIDE data. The evaluation of this contribution is addressed in SG 2.

Subgoal 2

Application study of the implementation (SG 1) based on the Body Comfort System (BCS) with regard to realisability.

We will evaluate our implementation of the UCM with a replication study. The study aims to replicate existing results of realisation analysis presented by Ochs [Och23], which will serve as Ground Truth (GT). For the evaluation, we instantiate our implemented UCM with an extension to the well known BCS[1]-Case Study (CS) and apply the realisation analysis method from Ochs [Och23]. By comparing our results with those of Ochs [Och23], we can draw conclusions about the expressiveness of our implementation.

Subgoal 3

Development of a concept for the integration of the SGEM into the UCM with regard to variability in time.

In SG3 we integrate concepts of managing variability in time from the SGEM into the UCM. Therefore we identify similarities between both modelling concepts by comparing their variability management mechanisms. The resulting unified model will describe PL-artefacts over time using concepts from the SGEM as additional attributes.

Subgoal 4

Further development of the BCS by adding evolution.

In SG4 we extend the BCS-CS from SG 2 by introducing several new system versions, that are intended to represent the evolution of the system over time. To this evolution we add SGE variation types to show that this SGE concept can be applied to other domains. We then implement the unified model from SG 3 and instantiate it with two versions of the extended BCS to show how accurate our model implementation can express variability in time.

Structure of the Thesis

The following thesis is structured as follows: Chapter 2 introduces the basic concepts that have been touched in this introduction. Chapter 3 covers the design choices

[1] https://github.com/TUBS-ISF/BCS-Case-Study-Full/

we made to extend the UCM with the goal of FM integration in Section 3.1, the integration of SGE concepts through time deltas in Section 3.2 and the unification of the integration together with the linking of the problem and solution space with mappings to allow the realisation analysis. Chapter 4 describes the implementation of our previously designed solution approaches, as well as the implementation of the tools required for the evaluation and the evolution of the BCS, where we added new versions and solution space artefacts. In Chapter 5 we evaluate and discuss our contributions. Chapter 6 gives an overview of related work. Chapter 7 summarises the thesis and gives an outlook on future work.

2. Basics

This chapter covers the basic terms and concepts of this thesis. Section 2.1 introduces basic concepts such as variability in time and space, problem space and solution space. Section 2.2 introduces the FM as a way of dealing with variability in space and Section 2.3 discusses the SGE approach as an approach of managing variability in time from a mechanical engineering perspective. Section 2.4 introduces deltas as another way of dealing with variability in time. Section 2.5 introduces the UCM as a model in which variability in space and time as well as problem and solution space can be combined.

2.1 Product Line Engineering

PLs, according to [NC13], provide an approach to mass customisation of (software) systems by constructing the system from reusable parts across a family of related products. This could be, as in our running example, a car PL where each customer can individually customise the car they want to order.

Apel et al. [ABKS13] distinguish product line engineering between requirements, known as the problem space, and artefacts that addresses those requirements, known as the solution space. Within this framework the problem space represents the different needs and constraints of user visible characteristics or behaviours of the product, called features, such as whether the car has a radio or not. The solution space comprises the artefacts and design decisions that make up the implementation of the PL, including code, components, architectures and configurations.

Variability refers to the ability of PLs to derive different products from a common set of artefacts [ABKS13]. The variability of a PL spans two dimensions, one in space, including product configurations and one in time, including the evolution of PLs over time. Variability in space includes product configurations that arise due to constraints between features. These constraints can be satisfied by selecting and deselecting features, resulting in various products, such as a car configuration with and without a radio. Variability in time occurs in a PL through updates, enhancements and adaption from one version to another, leading to products from different versions. Considering the dimensions of variability in time and space together enables a better understanding of how different product configurations are affected by temporal changes and contributes to better decisions in the development process.

2.2 Feature Model

The problem space describes the user visible characteristics or behaviours of the product, called features, and their dependencies. These features and dependencies can be described in a FM. According to Apel et al. [ABKS13], each feature can be referred to by its name. Each feature in a FM can be selected or not, leading to various configurations within a FM. This selection is limited by relations between features, groups of features or constraints that are also part of a FM.

A FM can be represented by propositional formulas, resulting from the dependencies between features, in conjunctive normal form (CNF) or by a graphical representation like the feature diagram (FD). In the FD features and dependencies are represented as a tree, where each node represents a feature and is labelled with the corresponding feature name. Those dependencies are distinguished in hierarchical tree constrains and cross-tree constrains. The hierarchical tree constrains are categorised in mandatory, optional, or-groups and alternative groups and give the tree its structure.

Tree Structure	Logic Term
mandatory feature f_i of parent p	$f \Leftrightarrow p$
optional feature f_i of parent p	$f \Rightarrow p$
or-group of features $f_1, f_2, \dots f_k$ of parent p	$\left(\bigvee_{i}^{k} f_i\right) \Leftrightarrow p$
alternative-group of features $f_1, f_2, \dots f_k$ of parent p	$\left(\left(\bigvee_{i}^{k} f_i\right) \Leftrightarrow p\right) \wedge \left(\bigwedge_{i,j,i\neg j}^{k} \neg (f_i \wedge f_j)\right)$

Table 2.1: Propositional Logic Representation of a FM [ABKS13].

The mandatory attribute of a feature is represented by a filled circle in the FD. In our running example, Figure 2.1, the feature `Finger Protection` is mandatory. This means the feature must be selected if the parent feature is selected, what corresponds to an equivalence in logic terms as seen in Table 2.1. If the feature is optional the feature can be selected if the parent feature is selected, which corresponds to an implication. An optional feature, in the FD, is represented by an unfilled circle on the feature, as feature `Power Window`, in Figure 2.1.

In an or-group at least one of the features must be selected if the parent feature is selected, as logic term, shown in Table 2.1, the or-group is represented by a disjunction over the features in the group which needs to be equivalent to the parent. In the FD an or-group is represented by a filled semicircle on the parent feature, as below feature `Infotainment` in Figure 2.1. If `Infotainment` is selected either `Radio` or `Navigation` or both need to be selected. An alternative-group is represented by an unfilled semicircle on the parent feature, as in the running-example below `Power Window`. Here exactly one of the features must be selected if the parent feature is

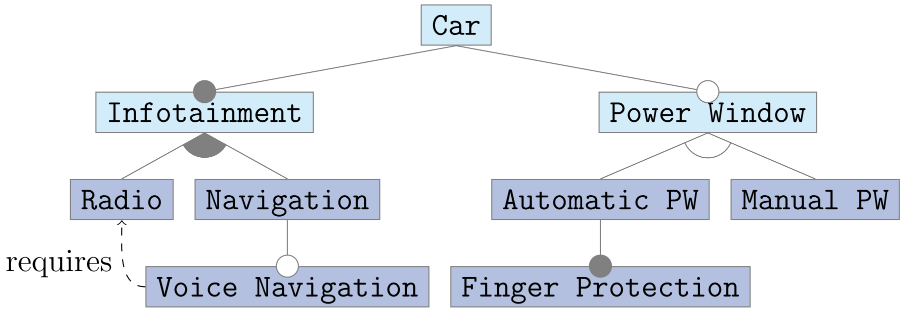


Figure 2.1: FD of the second version of the running example

selected. As a logic term an alternative group is an exclusive or over the group which needs to be equivalent to the parent.

Besides the hierarchical tree constrains, there are also constrains which do not fit into the structure of the tree. Those cross-tree constrains can either be represented by a labelled arrow as in the running example where the feature `Voice Navigation requires` the features `Radio`, or by logic term written below the FD.

2.3 System Generation Engineering

In the domain of mechanical engineering, SGE is used to describe the development of products in product generations. Albers [ABR17] defines SGE as the development of new product generations based on reference systems. The idea is that every concept added to a product has a reference system, either from another product that has already implemented the same concept, or as an evolution of an existing concept.

The SGE is used to describe generations through CVs, EVs and PVs. A generation G_{n+1}, with a previous generation G_n, can be understood as set of all variations $G_{n+1} = CV_{n+1} \cup EV_{n+1} \cup PV_{n+1}$ where CV_{n+1} is the set of CVs , EV_{n+1} is the set of EVs and PV_{n+1} the set of PVs.

As an example, we consider the development of a new generation of our car example to go through the three types of variation. The power window in the first generation (G_1) has a hand crank to move the window up and down manually. In the second generation (G_2) the window can be moved automatically by pressing a button. The system uses hydraulics to raise and lower the window in both the first and second generation, but is controlled once manually and once electrically.

Carryover Variations (CVs) reuse the reference system, changing only the interface for integration. In our example, this would be the window itself, which remains the same from the first to the second generation.

Embodiment Variations (EVs) change the shape of the subsystem but keep the principle of the solution. In the example this would be the hydraulics used to raise and lower the window. The principle of the hydraulics does not change but the shape changes because it needs to be controlled electrically.

Principle Variations (PVs) vary the solution principle by using new ones. A PV always goes with a EV. In the power window example, the hand crank that drives the hydraulics is replaced by a motor that can be controlled by a button, which is a new way of interacting with the power windows hydraulics.

These variation types allow us to calculate the complement of the degree of change over generations, where $\delta_{CV,n+1} = 100\%$ is an adaptive design and $\delta_{CV,n+1} = 0\%$ is a complete redesign of the system. $\delta_{CV,n+1}$ is calculated as follows

$$\delta_{CV,n+1} = \frac{|CV_{n+1}|}{|G_{n+1}|} = \frac{|CV_{n+1}|}{|CV_{n+1} \cup EV_{n+1} \cup PV_{n+1}|}$$

Applied to our example we would get a degree of change of 33.33% as only one of our three variation types were a CV, resulting in $\delta_{CV,2} = \frac{1}{3} = 33.33\%$.

2.4 Deltas

Deltas represent the specific changes or variations to accommodate different features, configurations or requirements based on a core model. With regards to PLs this means deltas are the element containing the change for a modeling element from one configuration to another or from one version to another.

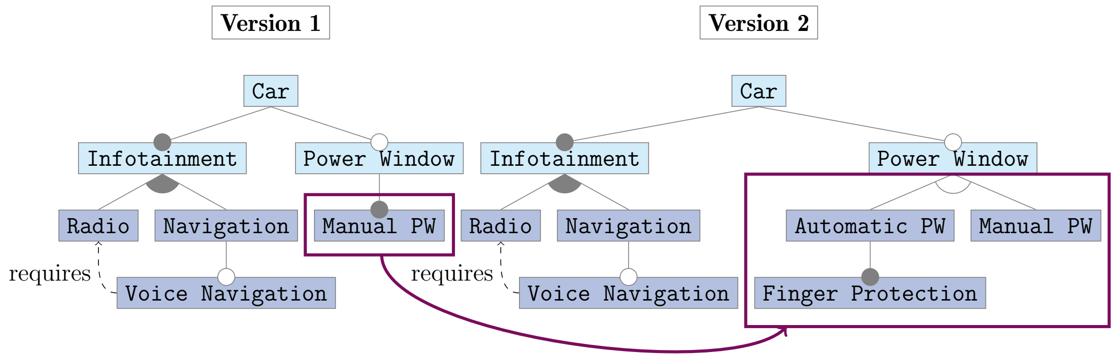


Figure 2.2: FD of the running example

In our running example in Figure 2.2 deltas would describe what changes from Version 1 to Version 2. The feature `Manual PW` will no longer be mandatory, but in an alternative-group. The feature `Automatic PW` is added to the same alternative-group and serves as parent of the new mandatory feature `Finger Protection`. The modification and the additions can be described by deltas.

Independent of FMs a general model M, according to Schaefer [Sch10], is a tuple $M = (E, R)$ of modeling elements E and a relation between those elements $R \subseteq E \times E$.

A Δ-Model over a model M is a tuple $\Delta = (\delta, Op)$, where δ is a constraint over E and $Op = \{op_1, op_2, \ldots, op_n\}$ is a set of delta operations over the model M which can either add, modify or remove an element e or add or remove a relation between two elements $r(e_1, e_2)$.

The Table 2.2 shows the delta operations Op of our running example. op_1 is a modification operation, that changes the mandatory tree constraint to an alternative-group. Then the two new features `Automatic PW` and `Finger Protection` are added as op_2 and op_3 and a new mandatory tree constraint is added as op_4. To the mandatory tree constraint a relation is added from `Automatic PW` as op_6 and a relation from the tree constraint to `Finger Protection` as op_7. This sub-tree

i	op_i	e_1	e_2
1	mod e_1	mandatory tree constraint above `Manual PW` to alternative-group	
2	add e_1	`Automatic PW`	
3	add e_1	`Finger Protection`	
4	add e_1	mandatory tree constraint	
5	add $r(e_1, e_2)$	alternative-group from op_1	`Automatic PW`
6	add $r(e_1, e_2)$	`Automatic PW`	mandatory tree constraint from op_3
7	add $r(e_1, e_2)$	mandatory tree constraint from op_3	`Finger Protection`

Table 2.2: Delta Operations of the running example

starting at `Automatic PW` is connected to the rest of the FD by a relation from the alternative-group from op_1 to `Automatic PW` as op_5.

The application of the operations creates a new model with, depending on the operation, a new element or dependency added, removed or in case of the modification operator first removed and then the modified one added. In our example the model would be Version 1 and after the deltas are applied the new model would be Version 2.

2.5 Unified Conceptual Model

The UCM was created to unify the problem and solution space in one model and also contain concepts for variability in space and time.

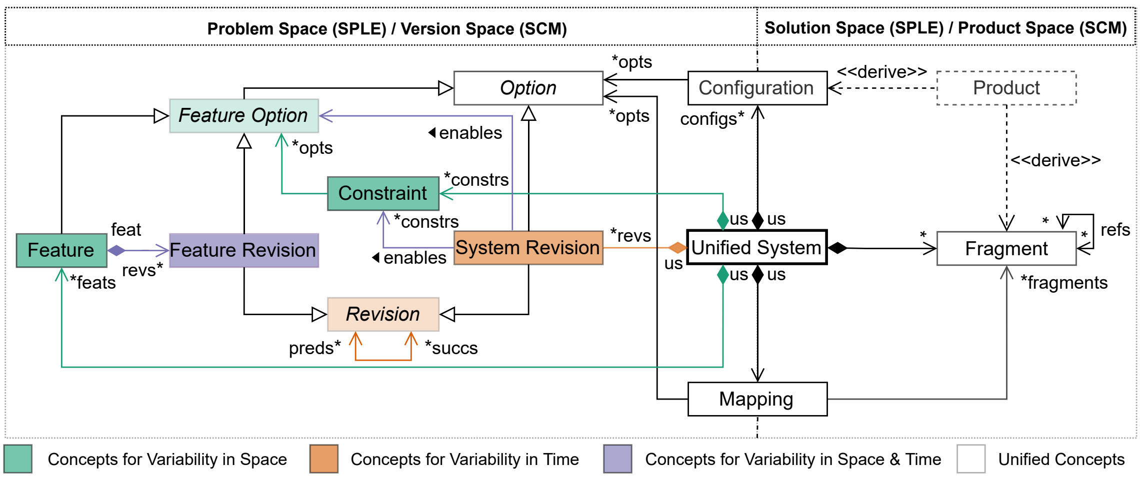


Figure 2.3: UCM according to Ananieva et al. [AGK+22]

Figure 2.3 shows the UCM as class diagram according to Ananieva et al. [AGK+22]. The model allows, similarly to the FM, to include features that are distinguished by their names. The dependencies between features are represented by the class `Constraint`. Each feature dependency is represented, similarly to the CNF, by a logical term contained in the `Constraint`. Instances of the class `System Revision` can be understood as versions of the unified system. They refer to other `System Revision` instances and to `Feature Revision` instances. The concepts of variability in space are part of the problem space. The solution space in the model contains products, fragments and classes such as `Configuration` and `Mapping` to connect the two spaces. The `Product` is derived from `Configuration` and `Fragment` where a configuration is a selection of `Options` and a `Fragment` is a concrete part of the product. Such a fragment can be lines of code or other model elements and gets mapped to an `Option` by `Mapping`. This ensures that each selectable `Option` has a corresponding counterpart in the solution space.

Wittler et al. [WKR22] and Ochs [Och23] have further developed the UCM, which is shown in Figure 2.4. The class `Product` is derived from a class `Configuration`, which refers to the problem space and `Components` of the solution space. The class `Component` can either be a `Software Component` and demand `Resources` or a `Hardware Component` and grant them. The class `Resource` consists of a value and a type. The class `Resourcetype` consists of a name, a unit of measurement, a boundary type, an the boolean attributes `isAdditive` and `isExclusive`. The boundary type can either be `LOWER`, `UPPER` or `EXACT`. `LOWER` states that a requested resource needs to be fulfilled with a provision that is greater than or equal to the requested one. `EXACT` requires that the provided resource value is equal to the requested one and `UPPER` requires the provided resource to be less or equal to the requested one. For example the response time type must be boundary type `LOWER`, because if a software component requires a response time of 12ms, a hardware component with a lower value such as 10ms will satisfy the demand, while a higher value such as 13ms will not. The attribute `isAdditive` determines whether or not it is allowed to add resources to satisfy a software components demands. For example, if a software component requires 100W of power and two hardware components each provide 70W, then the component's requirements can only be met if the addition of the resource is allowed. The attribute `isExclusive` determines whether its the resource that can satisfy the demands of multiple Software Components or not.

Our contribution mainly relies on the UCM by Ochs [Och23] but also takes parts of concepts considered by Ananieva et al. [AGK+22].

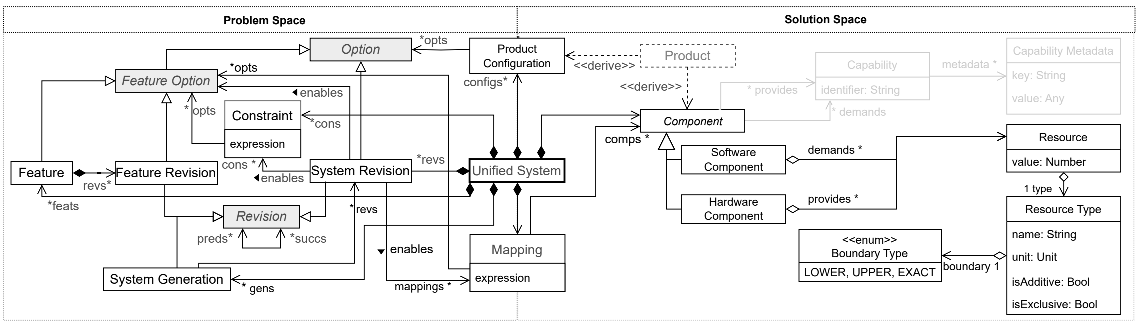


Figure 2.4: UCM according to Ochs [Och23]

3. Design

In this chapter the solution approaches are introduced. We extend the UCM to achieve the importation of FeatureIDE data that we addressed in SG1 in Section 3.1. We further extend the UCM to integrate the SGE and add evolution what we formulated in SG3. This solution approach takes place in Section 3.2. In Section 3.3 the design is unified and a specialised mapping is added to connect the problem and solution spaces.

3.1 Feature Model Integration

With respect to SG1, we modify the problem space of the UCM so that it represents a feature metamodel. Figure 3.1 shows an excerpt of the UCM with relevant parts concerning the representation of a feature meta-model. The following section describes the design decisions we made to achieve this goal.

A feature, according to the FM introduced in Section 2.2, has a name and can optionally have realisation artefacts in the solution space. Therefore, in the feature metamodel we represent a feature with a `name` and a boolean property `isAbstract`.

Apel et al. [ABKS13] proposed the FD as representation of the FM which differentiates between feature dependencies represented by tree hierarchy and by cross-tree constraints. Another representation of FMs would be through CNFs. The representation through CNFs would simplify the design effort, since we only need to design cross-tree constrains. But we not only want to represent FMs, but also want to enable the management of variability in space through our model. Therefore we adopt the differentiation of feature dependencies and introduce two specialisations of constraints to the feature meta-model: Hierarchical tree constraints and cross-tree constraints as detailed in the following.

Hierarchical tree constrains are described through hierarchy types as or-groups, alternative-groups, mandatory and optional, which where introduced in Section 2.2. Such a hierarchical tree constraint is always associated to exactly one parent feature and, depending on the hierarchy type, to at least one child feature. In our design, we distinguish between associated features and hierarchy types, as all hierarchical

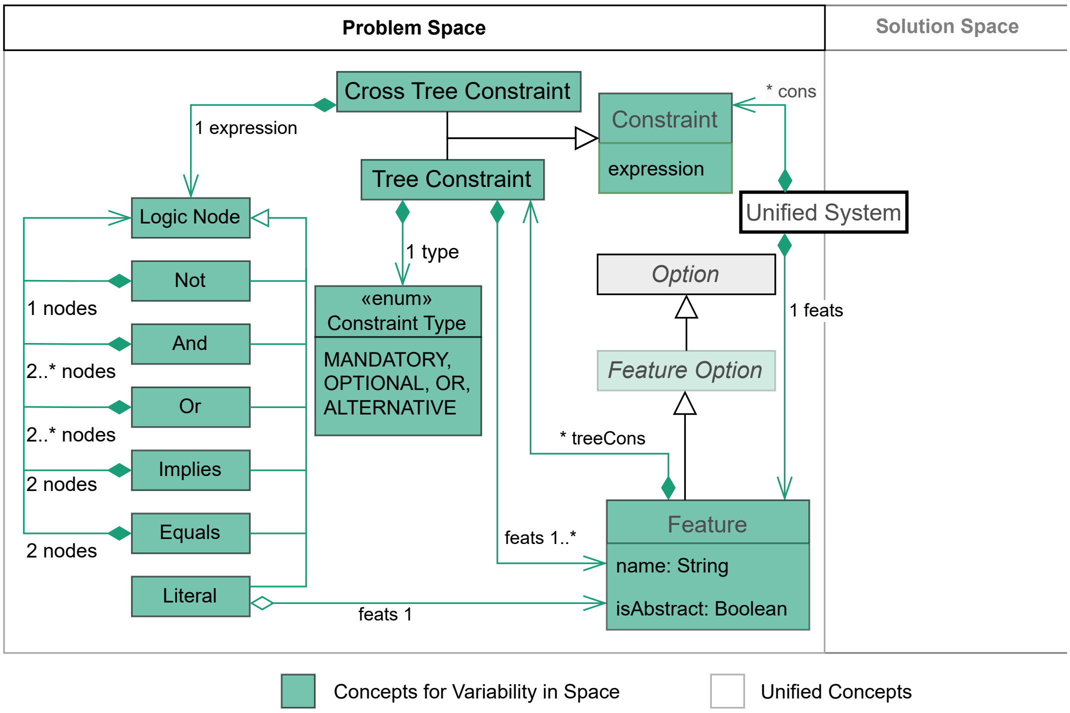


Figure 3.1: Extended UCM showing only the FM extensions

tree constraints share the characteristic of having associated features. This leads us to the class `Tree Constraint` which represents the hierarchical tree constraint and needs exactly one `Feature` as a parent and refers to at least one `Feature`. The type is represented by a `type` attribute, which is a reference to an enumerated class `Constraint Type` that represents the different hierarchy types, mandatory, optional, or, alternative, as names. This structure can be seen in Figure 3.1 represented by the associations between the classes `Feature`, `Tree Constraint` and `Constraint Type`.

A cross-tree constraint is a propositional logic term with features as literals. Therefore, we introduce an adapted model of a propositional logical formula as follows: A literal can be encapsulated by logical connectives such as not, and, or, implies and equals ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$). The type of logical connectives determines the number of outgoing nodes. A not node has only one child, implies and equals have two outgoing nodes and or and and have at least two outgoing nodes. A propositional logic term constructed in this way can then be encapsulated again by logical connectives, building up to a tree with a logical connective as root node. The tree constructed in this way is known as a concrete syntax tree [AA07] and can be used to represent propositional logic terms. It has the advantage that the order of evaluation of the term is ensured by traversing the concrete syntax trees and no brackets are required. We adapt the idea of concrete syntax trees in our design through the abstract class `Logic Node`, which represents the nodes of the tree. A node can be either a logical connective or a literal. We design this specialisation through specialised `Logic Nodes`. Each specialisation of the class `Logic Node` consists of a different number of children, which are references back to the class `Logic Node`. The class `Not` consists of one child, the class `Implies` and the class `Equals` consist of exactly two children, the class `And` and the class `Or` of at least two children and class `Literal` has no children but needs a reference to the class `Feature` instead. The class `Cross Tree`

`Constraint` represents the cross-tree constraints and consists of a reference to the class `Logic Node`as the propositional logic term. This design ensures the consistency of an expression even if, for example, the name of the feature changes.

Figure 3.2 shows the object diagram resulting from the parsed FD of Version 1 of our running example. The features of the FD are represented as instances of the class `Feature`. For example, the feature `Voice Navigation` is represented by `feats3` and is illustrated by the green arrow. You can see the structuring properties of the tree constraints in `treeCons0` and `treeCons1` where, as in the FD, the feature `Car` has two children with different hierarchy types. `treeCons2` represents an alternative-group and like the hierarchical tree constraint, has the parent feature `Infotainment` and two children. The red arrows show how the hierarchical tree constraints are mapped to instances of the class `Tree Constraint`. Finally, the `requires` cross tree constraint is represented by `cons0`, an instance of the class `Cross Tree Constraint`. A `requires` cross tree constraint can be understood as an implication and is therefore represented as such in the unified system by `expression0` as an instance of `Implies`. The structure of the concrete syntax tree can be seen in `expression0` by having exactly two children which are instances of `Literal` and each refer to their corresponding features.

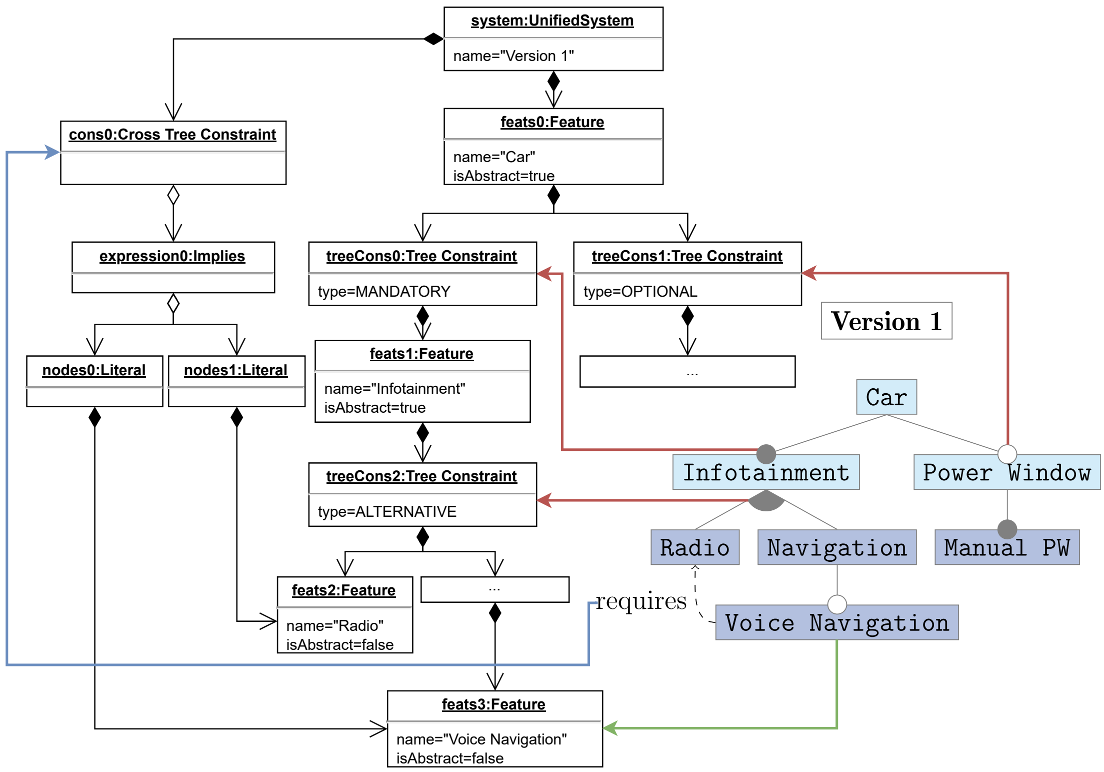


Figure 3.2: Representation of the FM of the first version of the running example as object diagram of the extended UCM

The object diagram clearly shows that no information is lost in the transformation from the FD to the unified system. The distinction between tree and cross-tree constrains plays an important role here, as without it the reconstruction of the structure would be almost impossible. This will allow us in the next chapter not only to parse the FD into a unified system, but also to parse a unified system back into a FD.

3.2 Integration of SGE Concepts through Time Deltas

This sections aims to achieve SG 3, where we formulated the goal of developing a concept to model the SGE by integrating the concepts of managing variability in time from the SGE into the UCM.

In Section 2.3 we introduced the term generation G_{n+1} that can be understood as set of all carryover, embodiment and principle variations $G_{n+1} = CV_{n+1} \cup EV_{n+1} \cup PV_{n+1}$ based on a predecessor generation G_n. For simplicity, we divide these variations into two categories: those that keep the (sub-)system as it is, which are CVs, and those that introduce changes, that are PVs and EVs. Consequently, a new generation can be conceptualised as the old (sub-)system with the applicable changing variations applied. This concept corresponds to the concept of deltas introduced in Section 2.4.

We modify the concept of Δ-models introduced in Section 2.4 to describe changes of the unified system over time. We understand an instance of the UCM as a model $M = (E, R)$ with the model elements E (e.g. instances of classes `Feature` and `Tree Constraint`) and the relations R between elements (e.g. the references of an instance of the class `Feature` to instances of the class `Tree Constraint`). In the according Δ-model $\Delta = (\delta, Op|_{R|_{Deltaable}})$ we restrict the delta operations Op to only be defined for $R|_{Deltaable}$. $R|_{Deltaable}$ is a subset of R and is defined in Equation 3.1. This allows us to store the Δ-Model as an ordered list of operations and elements of $R|_{deltaable}$, where the first element is a reference to an existing element in our model and the second element is either a new element or an existing element we wish to remove.

$$\begin{aligned} R|_{Deltaable} = &(\texttt{Feature} \times \texttt{Tree Constraint}) \\ &\cup (\texttt{Tree Constraint} \times \texttt{Feature}) \\ &\cup (\texttt{Software Component} \times \texttt{Resource}) \\ &\cup (\texttt{Unified System} \times (\texttt{Feature} \cup \texttt{Cross Tree Constraint} \\ &\cup \texttt{Mapping} \cup \texttt{Component} \cup \texttt{Resourcetype})) \end{aligned} \tag{3.1}$$

Figure 3.3 shows the changes we performed on the UCM to add this modified Δ-model. The Δ-model is represented by the class `Version` with an ordered list of references to `Delta` as the attribute `deltas`. As with delta modelling the `deltas` can be applied. In the class `Version` we have added the method `applyDeltas` for this purpose.

The class `Delta` represents the delta operations and has the attributes `type`, `position`, `value` and `valueRef`. The attribute `type` is a reference to `Delta Type`, which indicates whether the `Delta` adds, removes or modifies an element. The attribute `position` of the class `Delta`, has the same function as the first element of the relation tuple e_1 of $R|_{Deltaable}$ i.e. to point to the element that has or will have a reference to e_2 which is represented by `value` and `valueRef`. To ensure that (`position`, `value`) is in $R|_{deltaable}$ each class marked with an orange Δ in Figure 3.3 implements the interface `Delta Operations` and can be set as `position`. The type of `value` depends on the generic type of `Delta Operations` which must be the

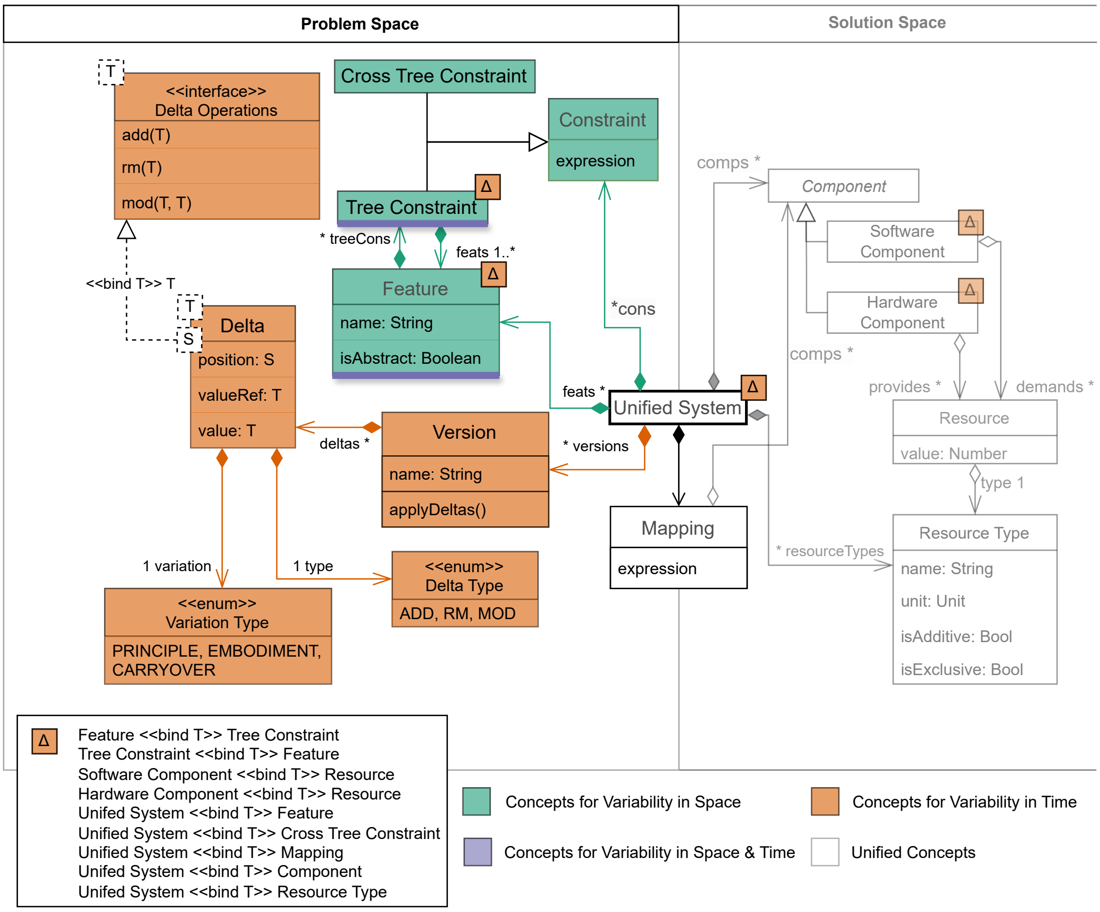


Figure 3.3: Extended UCM showing only the delta relevant part

same type as the generic type of `Delta`. The binding can be seen in the legend at the bottom right of Figure 3.3.

We integrate concepts of the SGE into delta-modelling by labelling each delta with a SGE variation type. We can consider the class `Delta` as variation and the class `Version` as generation, since it is made up of instances of the class `Delta`. To distinguish between the variation types of the class `Delta`, we add an attribute `variation` as a reference to a class `Variation Type`, which can be either `PRINCIPLE`, `EMBODY-MENT` or `CARRYOVER`, matching the variation types PV, EV and CV we introduced in Section 2.3.

We illustrate the functionality of deltas using the object diagram shown in Figure 3.4. The left-hand side of the object diagram shows Version 1 of the running example as described in Section 3.1 and the right-hand side shows the version with its delta. `delta0` modifies the instance of the class `Tree Constraint` referenced by `valueRef`, i.e. it deletes it and then adds a new instance of the same class referenced by `value` to the feature referred to by `position`. As the Figure shows, a delta can take an entire sub-tree and add it to the unified system, rather than just adding one element at a time as suggested by the delta operations in Table 2.2.

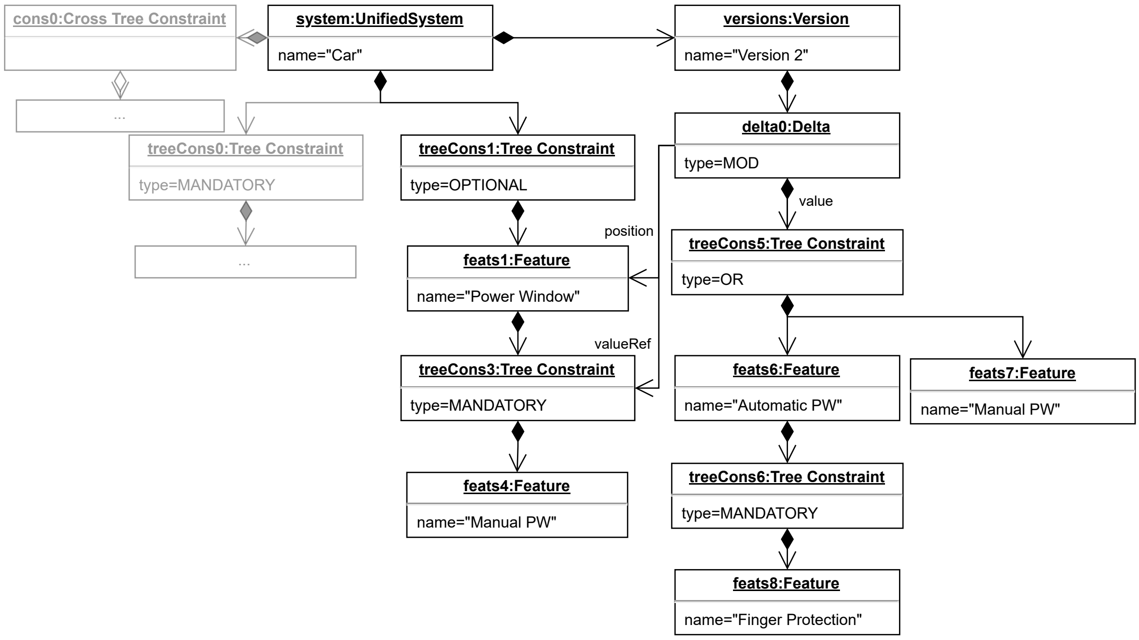


Figure 3.4: Representation of the FM of the first version of the running example with a delta to version two as object diagram of the extended UCM

3.3 Unifying Variability in Space and Time in Problem and Solution Space

We want to extend the concepts of managing variability in space (Section 3.1) and time (Section 3.2) beyond the boundaries of the problem space. Therefore, we specialise the class `Mapping`, which is used to connect the problem and solution spaces, by the class `Resource Mapping`. This class maps a feature to a resource demand, which consists of the value of the demanded resource, the resource type and the id of the software component in which the feature is implemented. We need to use the mapping and not the class `Software Component` because we still want to be able to manage variability. Therefore, the final resource demand of a software component depends on the version and configuration.

The resulting metamodel, which incorporates all our changes and allows for the management of variability in space and time in the problem and solution space, is shown in Figure 3.5. Appendix A.5 shows the full diagram.

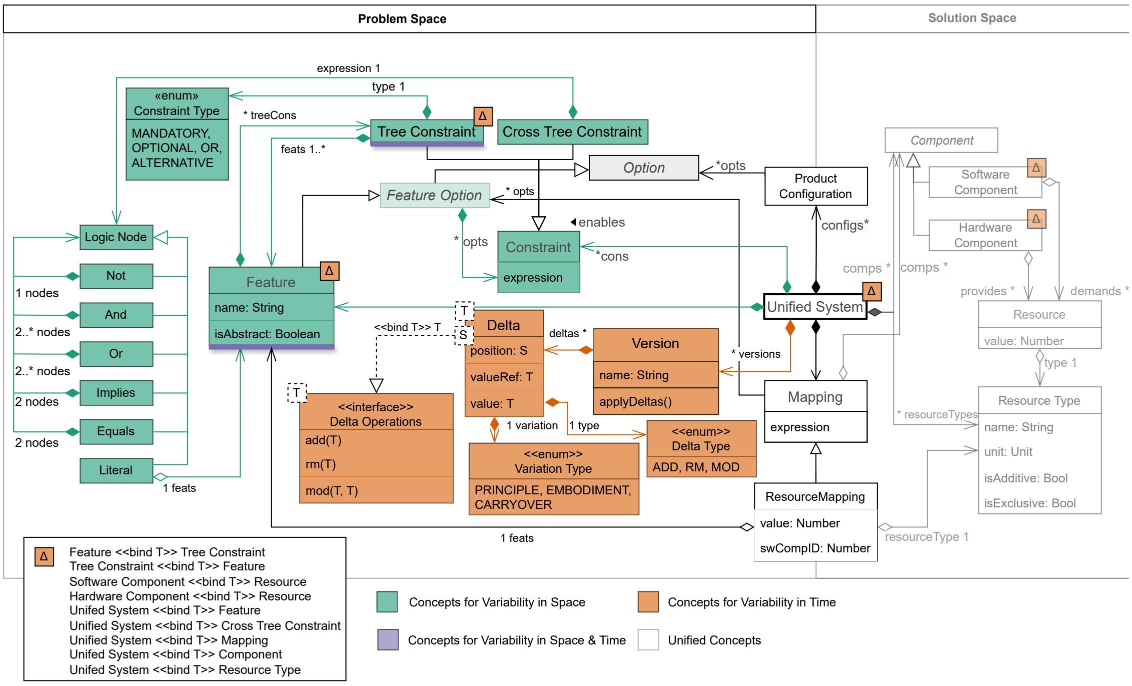


Figure 3.5: Extended UCM

4. Implementation

This chapter presents the code artefacts we created to implement and evaluate the UCM in Section 4.1 and the addition of resources to the various versions of the BCS and some new versions in Section 4.2.

4.1 Unified Conceptual Model Implementation

This section presents the code artefacts and tools used to implement and evaluate our extended version of the UCM.

4.1.1 Eclipse Modeling Framework

In SG1 we already specified that we are using the Eclipse Modeling Framework[1]. The EMF is used to create and edit models and then generate Java code based on the model. This model-centered view allows to concentrate on the design (Chapter 3), although Java has some limitations that made it necessary to adapt the design.

In Chapter 3 we designed the `Delta` with two generic types to ensure that deltas are only applied to relations that are elements of $R|_{Deltaable}$. The first generic S is the type of e_1, if we consider (e_1, e_2) as the relation tuple that is an element of $R|_{Deltaable}$ and the second generic T is the type of e_2. `Delta Operations` is an interface with a generic type T and the generic type S extends `Delta Operations` with the generic type T of `Delta` bound to the generic type T of `Delta Operations`. This ensures that classes that should be e_1, implement the interface `Delta Operations` with the generic T of `Delta Operations` bound to the type of e_2. Java does not allow the same interface to be instantiated with another generic class, as class `Unified System` needs to do, because, as we saw in Section 2.4, it implements `Delta Operations<Feature>`, `Delta Operations<Cross Tree Constraint>`, `Delta Operations<Mapping>`, `Delta Operations<Component>` and `Delta Operations<Resource Type>`. We can get around this limitation of Java by implementing the interface with the super type `Object` as `Delta Operations<Object>` and then distinguish in the methods `add` and `remove` the interface implements,

[1]https://eclipse.dev/modeling/emf/

which instance the argument passed is of. The method `modify` does not need to be touched since it calls the methods `add` and `remove` which then differentiate the instance.

For the serialisation of the unified system, we want the unified system to be build in such a way that when we serialise an instance of it, everything that is relevant to the instance is held by it. In Java we cannot specify whether a variable holds an object or just the reference to it. But EMF has this specification via the containment property. This property requires each object to be contained within another object to be able to save the objects properly. For the class `Delta` this means that we need to have two different attributes to refer to the value we want to either add or remove. When we want to remove a value we refer to it as `valueRef` as it is a reference to another object and is already contained in the `Unified System`. But if we want to add a value, the class `Delta` contains the value as the attribute `value`. Since an object cannot be contained in more than one object, the attribute `value` is removed when the delta is applied, because then the `value` is contained in the object referred to as `position`.

To be able to save and load an instance of the unified system and connect to other tools, we want to serialise the unified system. The EMF allows us to save and load the contents of the model to and from a Extensible Markup Language (XML) Metadata Interchange (XMI) [2] file. Listings 4.1 shows the XMI file containing the parsed UCM representation of Version 1 of our running example (Figure 3.1). The containment property of the EMF plays an important role here, as we want to serialise all references only through the unified system instance. Listing 4.1 shows this containment in the unified system by the fact that every element in the XMI file is at least one level below the unified system instance. An element consists of either a start-tag < *name* > and an end-tag < */name* > or, if there are no elements contained in between those tags, an empty-tag < *name/* >. Each element in the XMI file is named similarly to the name of the reference. For example the `Tree Constraint` is referenced as `treeconstraint` from the class `Feature` and so the element name is also `treeconstraint`. The attributes of the class, such as the `type` attribute of `Tree Constraint` are represented in the tags as XMI attributes. The attributes consist of a name-value pair of the form $name = value$. The name and value are similar to the name and value in the instance of the class in the unified system.

4.1.2 Adapters

FeatureIDE[3] is the state of the art tool for creating and displaying FDs. It is an a Eclipse plugin and we can use the libraries it provides for our adaptation.

Therefore we deserialise a FeatureIDE FD file with the libraries provided by the plugin. The adapter then translates the FD similar to what we have done with our running example in Figure 3.2. We extend this adapter further by allowing it to take into account the attributes in the FeatureIDE FD Ochs [Och23] uses to store the resource type mappings. To integrate the resource type mappings, we also need to add resource types to the model to give the mapping a resource to

[2] https://www.omg.org/spec/XMI/
[3] https://github.com/FeatureIDE/FeatureIDE

```xml
[...]
<ucm:UnifiedSystem [...] name="Car">
  <feats name="Car" isAbstract="true">
    <treeconstraint type="MANDATORY">
      <feature name="Infotainment" isAbstract="true">
        <treeconstraint type="OR">
          <feature name="Radio" isAbstract="false"/>
          <feature name="Navigation" isAbstract="false">
            <treeconstraint type="OPTIONAL">
              <feature name="Voice Navigation" isAbstract="false"/>
            </treeconstraint>
          </feature>
        </treeconstraint>
      </feature>
    </treeconstraint>
    <treeconstraint type="OPTIONAL">
      <feature name="Power Window" isAbstract="true">
        <treeconstraint type="MANDATORY">
          <feature name="Manual PW" isAbstract="false"/>
        </treeconstraint>
      </feature>
    </treeconstraint>
  </feats>
  <cons id="0">
    <expression xsi:type="ucm:Implies">
      <nodes xsi:type="ucm:Literal" featureoption="//@feats.0/
          @treeconstraint.0/@feature.0/@treeconstraint.0/@feature.1/
          @treeconstraint.0/@feature.0"/>
      <nodes xsi:type="ucm:Literal" featureoption="//@feats.0/
          @treeconstraint.0/@feature.0/@treeconstraint.0/@feature.0"/>
    </expression>
  </cons>
</ucm:UnifiedSystem>
```

Listing 4.1: Parsed UCM representation of Version 1 of the running-example

reference. So we use the opencsv[4] library to deserialise the resource types stored in a comma-separated values (CSV) file. The same deserialiser can be used to read the CSV file in which Ochs [Och23] has stored the resource provisions of the hardware components.

In the CSV resource type file, each line represents a resource type. Each line consists of four values, the first one being the ID of the resource, the second the binary value `isAdditive`, the third the binary value `isExclusive` and the last value is `boundary` where the three possible values are represented by the numbers `LOWER` : 0, `UPPER` : 1 and `EXACT` : 2.

In the resource provisioning file, each row represents a hardware component and each column represents the resource type whose ID matches the corresponding column number when numbered consecutively from zero in ascending order. The value in each cell then represents the quantity or value of the corresponding resource type provided by the hardware component.

[4] https://opencsv.sourceforge.net/

The resource demands of a feature are represented by its attributes in the FeatureIDE FD. The name of the attribute consists of a tuple (i, j), where i is the ID of the software component and j is the ID of the resource type. The value of the attribute is the demanded number or value of the resource.

We also wrote an adapter the other way around in order to get a FeatureIDE FD out from a unified system instance. So we read the UCM XMI file and deserialise it with the tools provided by the EMF to a unified system instance. After adapting, the FeatureIDE FD instance is serialised using the libraries provided by FeatureIDE.

4.1.3 Combined Problem Solver

The combined problem solver tool was implemented by Ochs [Och23] to calculate all realisable and valid configurations of a FeatureIDE FD with corresponding resource types, and resources provisions and a CNF representation of the FM. We modified his tool to instead take our XMI representation of the unified system to calculate all valid and realisable configurations. We later use this tool to calculate the valid and realisable configurations using the unified system representation for evaluation purposes and to replicate the results of Ochs [Och23] achieved using the unmodified combined problem solver tool. Therefore we need to make the following adjustments:

Extract the resource types from `Resourcetypes`

The resource types are extracted from the serialised unified system instead of the resource type CSV file Ochs [Och23] used.

Extract the feature demands from `Mappings` and the resource provisions from `Hardware Components`

To extract the corresponding data from the mappings and cross-tree constraints we need to resolve the references in the XMI file, as in line 26 of Listings 4.1. The references are built as concatenations of elements of the tree. In the example in line 26 *`//@feats.0`* stands for the first feats element of the root, the `ucm`, element. The `/` stands for going into the selected element and the `treeconstraint.0` stands for the first `treeconstraint` element below the current element.

Calculate the a CNF of the FM from `Cross Tree Constraints` and `Tree Constraints`.

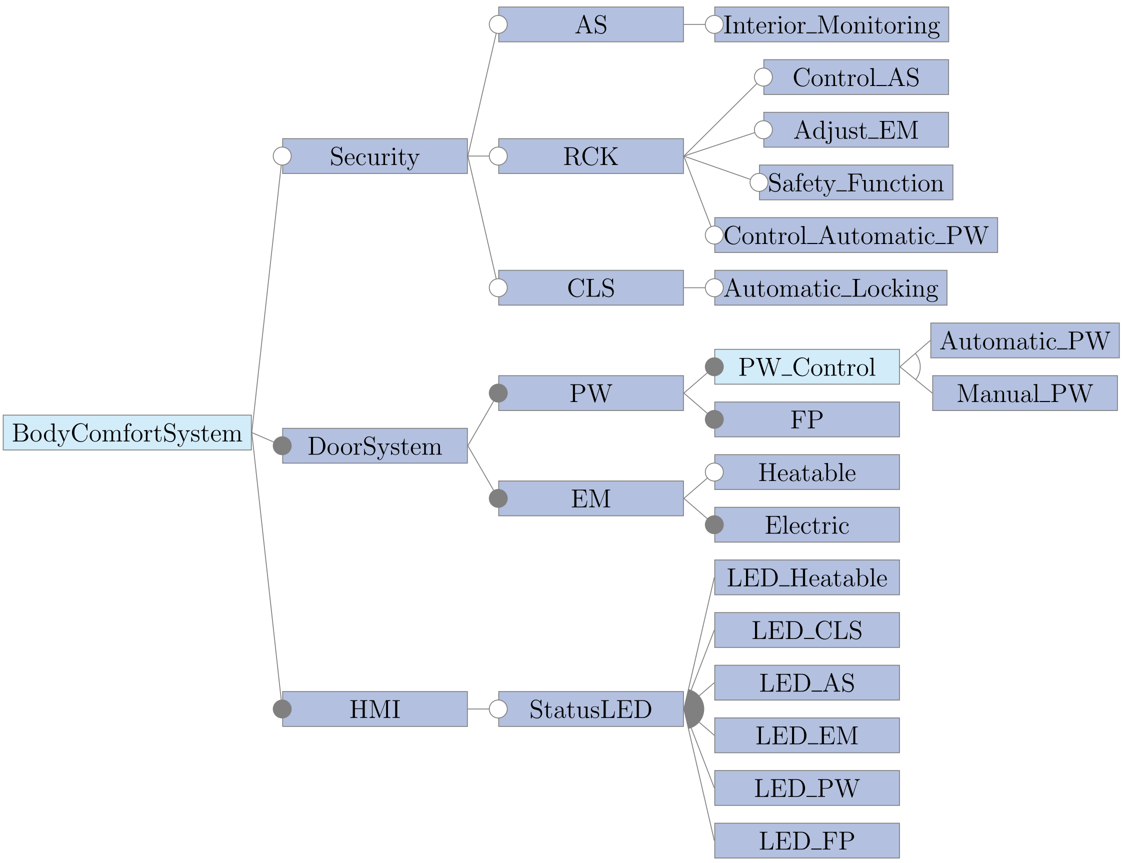
The CNF is constructed by conjuncting the logical term representation of tree constraints and cross-tree constraints. The cross-tree constraints can be transformed into CNF by traversing each concrete syntax tree that presents each cross-tree constraint. The logic terms of a tree-constraint are looked up by the type of the tree-constraint in Table 2.1. And then build according to the logic term with the features in the tree-constraint and the parent feature.

4.2 Evolution of the Body Comfort System

This section describes how the existing evolution of the BCS is extended by adding more versions, resource provisions and demands, and how evolution can be classified through SGE operations.

Version 1.0

The base version of the BCS Version 1.0 consists of a 27 features which split up in tree subsystems `Security`, `DoorSystem` and `HMI`, as you can see in the FM in Figure 4.1. The feature `DoorSystem` has electric adjustable Exterior Mirrors (EMs) that optional can be heatable and Power Windows (PW) can be either automatic or manually controlled and need to have a Finger Protection (FP). The feature `Security` is optional and consists of functionality for Alarm System (AS), Remote Control Key (RCK) or Central Locking System (CLS). The feature `HMI` consists of multiple `StatusLED`s which can be chosen individually via an or-group representing status and signal lights. There are six cross-tree constraints in the FM of the BCS CS.



`LED_AS` $\Rightarrow$ `AS`

`LED_Heatable` $\Rightarrow$ `Heatable`

`Control_Automatic` $\Rightarrow \neg$ `Manual_PW`

`LED_CLS` $\Rightarrow$ `CLS`

`RCK` $\Rightarrow$ `CLS`

`Control_AS` $\Rightarrow$ `AS`

Figure 4.1: FM of BCS

ID	isAdditive	isExclusive	boundary	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 4.1: Resource Types of Version 1.0

Hardware Component (hw_j)	Provisions (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)	$rp_1^2 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$

Table 4.2: Resource Provisions of Version 1.0

Feature	Software Component sw_i	Resource Demands rd_k^i
LED_AS	sw_0	$rd_0^0 = 2$
LED_FP	sw_0	$rd_0^0 = 1$
LED_CL	sw_0	$rd_0^0 = 1$
LED_PW	sw_0	$rd_0^0 = 1$
LED_EM	sw_0	$rd_0^0 = 1$
LED_Heatable	sw_0	$rd_0^0 = 1$
Electric	sw_1	$rd_1^7 = 10$ $rd_1^1 = 5$
Heatable	sw_1	$rd_1^1 = 20$
FP	sw_2	$rd_6^2 = 1$
Manual_PW	sw_2	$rd_6^2 = 2$
Automatic_PW	sw_2	$rd_7^2 = 5$ $rd_6^2 = 3$
RCK	sw_3	$rd_2^3 = 2$ $rd_3^3 = 10$
Control_Automatic_PW	sw_1	$rd_7^1 = 5$
	sw_3	$rd_2^3 = 2$
Adjust_EM	sw_2	$rd_7^2 = 10$
	sw_3	$rd_2^3 = 2$
Control_AS	sw_3	$rd_2^3 = 2$ $rd_2^3 = 2$
Safety_Function	sw_3	$rd_2^3 = 2$
AS	sw_3	$rd_3^3 = 100$ $rd_5^3 = 4$
Interior_Monitoring	sw_3	$rd_3^3 = 700$ $rd_5^3 = 1$
CLS	sw_3	$rd_3^3 = 10$
Automatic_Locking	sw_3	$rd_4^3 = 1$

Table 4.3: Resource Demands of Version 1.0

In the solution space Ochs [Och23] added four software components. The `hmi-controller` (sw_0) controls the Human Machine Interface (HMI) and all `StatusLEDs`. The `power-window-controller` (sw_1) controls the `PW` and `exterior-mirror-controller` (sw_2) the `EM`. At last the `security-controller` (sw_3) controls the `Security` subtree.

In Table 4.1 the resource types are specified. As introduced in Section 2.5, a resource type has an identifier k and the three resource type properties `isAdditive`, `isExclusive` and `boundary`.

The resource demands can be found in Table 4.3. The resources are provided by the hardware components in Table 4.2 where each hardware component provides an amount or number of resources.

The resource provisions are designed to not fulfil all resource demands thus not all valid configurations are also realisable. In this version the resource demands of the feature `Interior_Monitoring`, $rd_5^3 = 1$, can not be fulfilled. The resource type r_5 describing the number of Security Video Cameras has an `EXACT` boundary. In the resource provisions only hw_1 provides $rp_5^1 = 4$ which does not match the request exactly. This means that out of 11616 valid configurations only 6528 different configurations are realisable.

Version 1.1

Version 1.1 makes changes of the architecture because of new passenger safety regulations. The new safety regulations require a dedicated hardware, to separate security and safety. For this purpose hardware `safety-hardware` is added and takes over all resource provisions related to passenger safety.

Appendix A.1.2 shows the details in tables.

Version 2.0

In Version 2.0 Figure 4.2 shows, that the subsystem rooted in feature `Wiper` is added as a mandatory feature. The feature `Wiper` needs either a `High_Quality_Sensor` or a `Low_Quality_Sensor` to detect rain and a `High_Quality_Wiper` or a `Low_Quality_Wiper` to remove rain from the windshield. The low quality wiper can only be on or off and the low quality wiper only detects rain or not, while the high quality features can be more granular. Optionally a `Clean` feature can be chosen to clean the windshield.

The extension of the `BodyComfortSystem` through `Wiper` is a PV. All the features that are PVs have a green border in Figure 4.2. The `StatusLED` Feature itself is only an EV. All the feature that are EVs have an orange border in Figure 4.2. Everything else is not changed and thus are CVs.

From this example we can generalise rules to classify the variation types in FDs. A feature that is adding a new concept or solution principle to the system can be considered as PV. Parent features of such a PV are at least an EV since their shape needs to change if a new feature is added. Only if the variation is not present or minimised to only a change of parameters the variation is a CV.

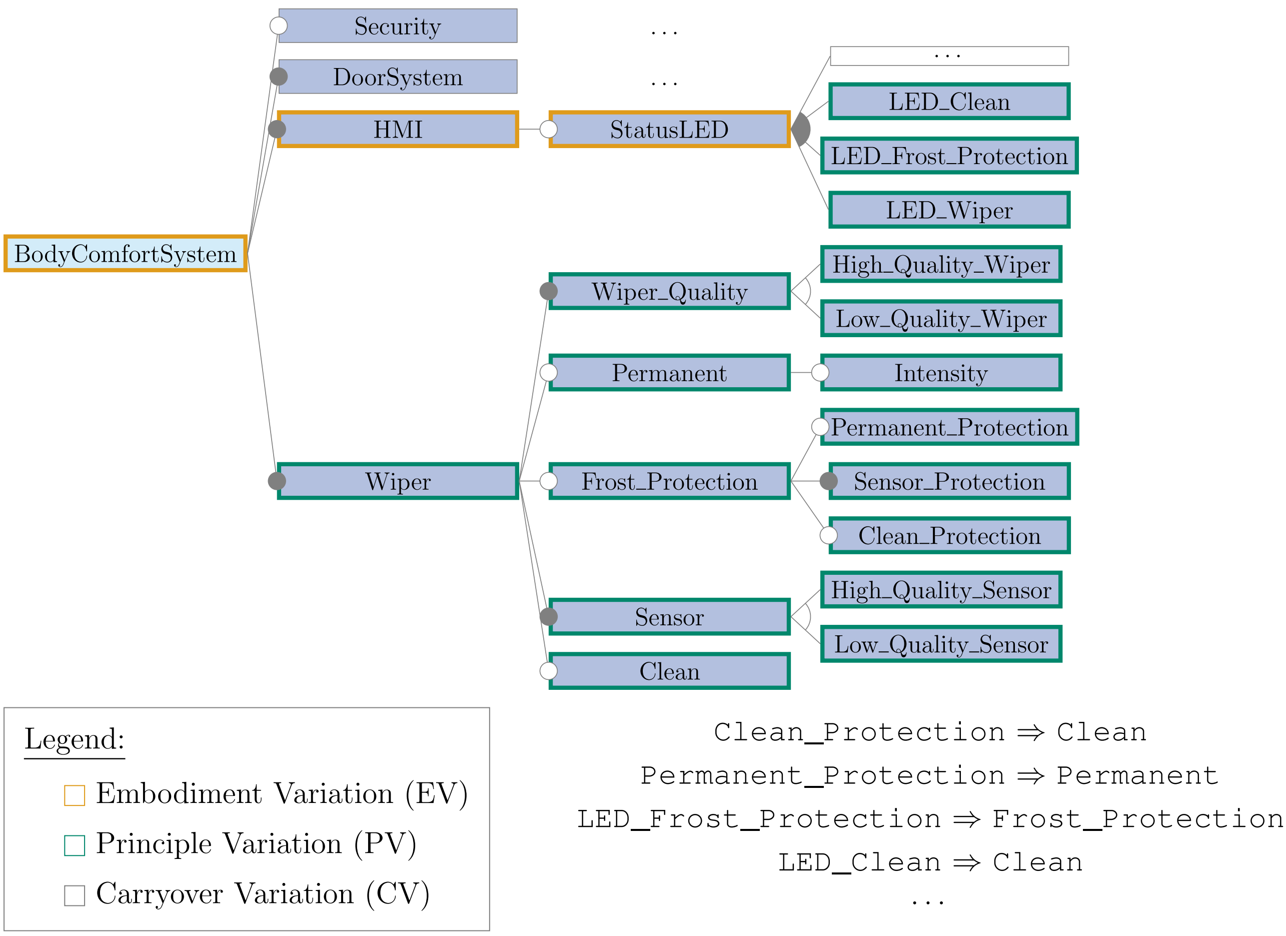


Figure 4.2: FM of BCS Version 2.0

ID	isAdditive	isExclusive	boundary	Unit	Description
8	`True`	`True`	`LOWER`		no. Pumps
9	`True`	`False`	`LOWER`		no. Temperature Sensors
10	`True`	`False`	`LOWER`		no. Liquid Level Sensor

Table 4.4: Resource Types added in Version 2.0

Feature	Software Component sw_i	Resource Demands rd_k^i
LED_Wiper	sw_0	$rd_0^0 = 1$
LED_Frost_Protection	sw_0	$rd_0^0 = 1$
LED_Clean	sw_0	$rd_0^0 = 1$
Clean	sw_4	$rd_8^4 = 1$ $rd_{10}^4 = 1$
Frost_Protection	sw_4	$rd_9^4 = 1$
Low_Quality_Wiper	sw_4	$rd_1^4 = 10$ $rd_7^4 = 5$
High_Quality_Wiper	sw_4	$rd_1^4 = 10$ $rd_7^4 = 10$
Low_Quality_Sensor	sw_4	$rd_7^4 = 5$
High_Quality_Sensor	sw_4	$rd_7^4 = 10$

Table 4.5: Resource Demands added in Version 2.0

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 35$ $rp_7^0 = 16$
`security-hardware`(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$ $rp_8^2 = 1$ $rp_9^2 = 1$ $rp_{10}^2 = 1$

Table 4.6: Resource Provisions of Version 2.0

As software component the `wiper-controller` (sw_4) is added to control the wiper subsystem. The `Wiper` feature can be seen as a feature for passenger safety because the driver can see better with a clean windshield. This is the reason why the required resource demands of Table 4.5, except for the LEDs, are provided by the `safety-hardware` as you can see in Table 4.6.

The model has nine `StatusLEDs` but `safety-hardware` only provides eight. Therefore not every configuration is buildable.

Version 3.0

Version 3.0 adds electric seat adjustment with memory function on the driver's key. Therefore, a new feature Seat is added as optional child of the feature BodyComfort-System and the resource types number of motors and long-term memory are added. These are demanded by a new software component seat-controller (sw_5) and get fulfilled by the infotainment-hardware.

Appendix A.1.4 shows the details in tables and diagrams.

Version 3.1

It was discovered that there was no hardware component that satisfies the demand of exactly one security camera for Interior_Monitoring, why in this Version security-hardware-2 was added which fulfills the demand of the feature Interior_Monitoring. In addition, demands for the response time to the finger protection functionality are added.

Appendix A.1.5 shows the details in tables.

Version 4.0

In Version 4.0 a new feature Windows_Heatable is added as an optional child of BodyComfortSystem with its own StatusLED. A new software component windows-heat-controller (sw_6) is added, which demands the use of a temperature sensor that is already included by the wiper update.

Appendix A.1.6 shows the details in tables and diagrams.

Version 5.0

In Version 5.0 a new feature Automatic_Headlights is added as an optional child of BodyComfortSystemand the resource types number of parking lights, number of daytime running lights, number of low beams, number of high beams, number of ambient light sensors and number of front proximity sensors are added. These resource types are demanded by a new software component, the headlight-controller (sw_7), and provided by a new hardware component, headlight-hardware.

Appendix A.1.7 shows the details in tables and diagrams.

Version 5.1

In response to customer feedback requesting more granular control over lighting options, Automatic_Headlights gets divided into three distinct choices: Beam, Parking_Lights, and Daytime_Running_Lights. Instead of a binary decision of whether to have automatic headlights or not, customers can now select from a range of automatic lighting options, tailoring their vehicle's lighting setup to better suit their needs and preferences.

Now the corresponding resource types can be removed and only number of ambient light sensors and number of front proximity sensors are kept from the previous version but with another ID. Also the headlight-controller (sw_7) now requests a certain Power Specification to light up or dim the different headlight features. The headlight-hardware now needs to provide a lot of Power instead of the Lights.

Appendix A.1.8 shows the details in tables and diagrams.

5. Evaluation

In this chapter we evaluate the implementation of the UCM described in Chapter 3 (SG 1 & 2) and the SGE described in Chapter 3 through the extended BCS-CS implemented in Chapter 4 (SG 4). For the evaluation of our SGs we derive the following Research Questions (RQs):

RQ 1: Can the UCM be used to reproduce realisation analysis?

With RQ 1 we evaluate the first SG which aims to unify and simplify existing tools and considerations as the four file solution of Ochs [Och23] for realisation analysis using FDs with corresponding resource provisions and demands. We will evaluate this goal, by replicating the results of Ochs [Och23] (SG2). Therefore, we compare the sets of valid and realisable configurations of two CSs, once calculated from the four files and once calculated from our one file solution. We expect the sets to match exactly and then we can consider the implementation of the UCM as complete and accurate in the parts the CSs are covering.

RQ 2: Can concepts of delta modelling be used to express variability in time in the extended UCM?

With RQ 2 we evaluate the third SG which aims to include concepts of the SGE to manage variability in time and transfer these concepts to the UCM using time deltas. These deltas can express the SGE variation types. We will evaluate this goal by verifying the capability to manage variability in time of our implementation with the BCS-CS evolution we created in SG 4. We consider the sets of valid and realisable configurations of two successive versions of the BCS-CS as GT. To calculate the GT we use the approach provided by Ochs [Och23]. We then use our approach to identify the valid and realisable configurations for the CS evolution an compare our results to the GT. We expect the sets to match exactly and then we can consider the implementation of the UCM as able to express variability in time. We also calculate the degree of change, a metric to express how much of the system changes from one system generation to another, for the BCS evolution to show that the concepts of variation types of the SGE can be used for cross-domain PLs.

5.1 Experiment Setup

5.1.1 Research Question 1

Ochs [Och23] used four files to compute all the valid and realisable configurations, whereas our unified approach requires only one file containing the instance of the UCM. This instance is instantiated with the FM, the resource provisions and the resource types. The unified system model was developed in version 2.36.0.v20231107-0612 of the EMF using Java version 1.8. The combined problem solver used to compute the GT and the modified version of the combined problem solver, which takes the serialised unified system as input and computes our results, were run in Python 1.10.

Ochs [Och23] used two CSs, CS 1 and the BCS-CS, to evaluate his results. The first CS, CS 1, is used to maximize the coverage of the developed concepts and the BCS-CS to show that his work also scales to a typically used case study. The CSs are explained in more detail below:

Case Study 1

Even though we only need one file, we will spread the explanation of CS 1 over several representations for better clarity as we did before in Section 4.2 for the BCS-CS.

CS 1 covers a car PL that aims to cover the twelve possible characteristics of resource types. Table 5.1 shows these characteristics, which are the combination of the different values of the attributes `isAdditive`, `isExclusive` and `boundary`. The meaning of these attributes was discussed in Section 2.5.

The FD of the CS includes 13 features and no cross-tree constraints, as shown in Figure 5.1. The customer can choose between a big and a small infotainment system and whether or not he wants a tyre-pressure monitor and electric adjustable seats. If the customer decides to have electric adjustable seats there can be chosen between a 6-way and a 10-way seat adjustment and between front and back or only front motorization. Mandatory for all configurations is the monitoring for critical components. The allocation of resources to the features together with a software component is shown in Table 5.3 and the resource provisions are shown in Table 5.3.

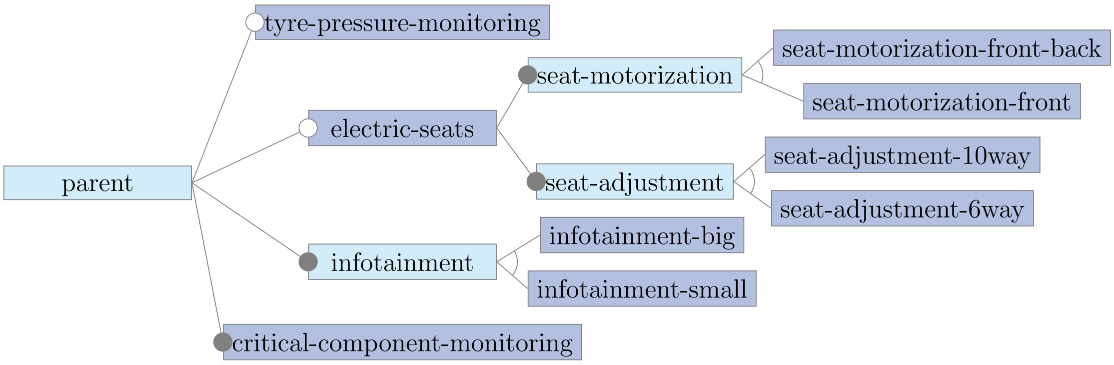


Figure 5.1: FM of CS1

ID	isAdditive	isExclusive	boundary	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 5.1: Resource Types of CS1

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware` (hw_0)	$rp_0^0 = 15$ $rp_1^0 = 50$ $rp_2^0 = 3$ $rp_6^0 = 8$ $rp_7^0 = 1$
`ccm-hardware` (hw_1)	$rp_3^1 = 5$ $rp_4^1 = 1$ $rp_6^1 = 4$ $rp_8^1 = 1$
`car-periphery-hardware` (hw_2)	$rp_5^2 = 1$ $rp_9^2 = 6$ $rp_{10}^2 = 2$ $rp_{11}^2 = 4$

Table 5.2: Resource Provisions of CS1

Body Comfort System-Case Study Version 1.0

We use Version 1.0 of the BCS evolution history, which we described in detail in Section 4.2.

Ochs [Och23] used a modified version of the BCS-CS. Instead of the cross-tree constraint `Control_Automatic` $\Rightarrow \neg$ `Manual_PW` seen in Figure 4.1 he used `Control_Automatic` $\Leftrightarrow \neg$ `Manual_PW`. This modification results in less valid and realisable configurations, because an equation is more stringent than an implication.

To obtain the corrected GT consisting of the set of valid and realisable configurations we ran the combined problem solver of Ochs [Och23] with the corrected data and collected the results as GT.

Feature	Software Component sw_i	Resource Demands rd_k^i
`infotainment-small`	sw_0	$rd_0^0 = 5$ $rd_1^0 = 100$ $rd_2^0 = 1$ $rd_6^0 = 2$ $rd_7^0 = 1$
`infotainment-big`	sw_0	$rd_0^0 = 10$ $rd_1^0 = 100$ $rd_2^0 = 3$ $rd_6^0 = 6$ $rd_7^0 = 3$
`critical-component-monitoring`	sw_1	$rd_4^1 = 1$ $rd_6^1 = 2$ $rd_8^1 = 1$
	sw_2	$rd_3^2 = 2$ $rd_6^2 = 1$
`electric-seats`	sw_3	$rd_5^3 = 1$
`seat-adjustment-6way`	sw_3	$rd_9^3 = 6$
`seat-adjustment-10way`	sw_3	$rd_9^3 = 10$
`seat-motorisation-front`	sw_3	$rd_{10}^3 = 2$
`seat-motorisation-front-back`	sw_3	$rd_{10}^3 = 4$
`tyre-pressure-monitoring`	sw_4	$rd_{11}^4 = 4$

Table 5.3: Resource Demands of CS1

5.1.2 Research Question 2

To show that the deltas in the UCM are working as intended we create two serialised instances of the unified system to compare them to our GT. The first instance consists of Version 1.0 of the BCS-CS and contains deltas representing what changes from Version 1.0 to Version 2.0, which adds the wiper system. The second instance consists of the first instance but the deltas are applied. We take the tool of Ochs [Och23] to create the GT consisting of the set of valid and realisable configurations for Version 1.0 and Version 2.0. For the evaluation we compare the sets of valid and realisable configurations of Version 1.0 (GT) with the unified system instance with unapplied deltas and the sets of valid and realisable configurations of Version 2.0 (GT) with the applied deltas instance. If the sets of realisable configurations are the same in each case we assume that our delta model works as intended.

Besides the verification of the integration of deltas we want to show that SGE variation types can be applied to models outside the domain of mechanical engineering. Therefore, we calculate the complement of the degree of change, of the FDs of the BCS evolution we implemented in Chapter 4. We introduced the complement of the degree of change in Section 2.3 as the share of CVs, $\delta CV, i$. Based on the size of the change, we expect a larger (more than 20%) or smaller (less than 20%) value for the degree of change. This expectation will serve as GT for the comparison. For Version 1.0 we expect the degree of change to be at 100% because the whole system is new. For Version 1.1 we expect the degree of change to be 0% because no features are modified. For Version 2.0 we expect a high degree of change because many features are added, where the change to Version 3.0 is very small. The difference from Version 3.0 to Version 3.1 is expected to be 0% as no features are modified. The degree of change for Version 4.0 is expected to be low, but higher than of Version 3.0 and the degree of change of Version 5.0 is expected to be as low as the degree of change of Version 3.0. The degree of change of Version 5.1 is expected to be higher than the degree of change of Version 5.0 but lower than of Version 2.0.

5.2 Experiment Execution

For the first two experiments we need to calculate the valid and realisable configurations that will serve as GT and our results. The GT is calculated with the combined problem solver and our results are calculated with the modified combined problem solver, taking as input the unified system instances representing the CSs. To compare these sets we need to introduce a compare operator. According to Ochs [Och23] two configuration sets (CS) are equal, $A \equiv_{CS} B$, if the cardinality of the sets are equal and for each configurations from the first set A there needs to be the same configuration in the second set B. A configuration is equal to another if both configurations have the same features selected.

For the GT of the first experiment we run the combined problem solver of Ochs [Och23] with the FD and the CNF representation of the CS provided by the FeatureIDE and the CSV files representing the resource types and resource provisions of the CSs. The files are provided by Ochs [Och23]. The files can also be rebuilt based on the figure and tables that describe each CS. To calculate the valid and realisable configurations with the new method we first parse the files used to calculate the GT into a unified system instance with the adapter tool we wrote and then serialise this instance to a file with the unified system serialise tool. The serialised file is processed by the modified version of the combined problem solver, that calculates our result. The GT and our results are then compared by the evaluation tool we wrote.

For the second experiment we calculated the set of valid and realisable configurations that build the GT for Version 1.0 and Version 2.0 of the BCS-CS with the combined problem solver. To calculate the GT for Version 1.0 we can use the same files as before, and for Version 2.0 we obtain the necessary files by creating them based on the figure and tables of the CS described in Section 4.2. To create the unified system instance with deltas we take the unified system of Version 1.0 of the BCS and add deltas describing the added and modified features, mappings, resource types and resource provisions. We then serialise the unified system instance and apply the deltas we added to serialise the new state of the unified system instance. We calculate the sets of valid and realisable configurations based on the two unified system instances as in the experiment before. We then compare the set of valid and realisable configurations obtained from the unified system instance with the unapplied deltas with Version 1.0 of the GT and the set of valid and realisable configurations obtained from the unified system instance with applied deltas with Version 2.0 of the GT.

In the last experiment we calculated the degree of change or more specifically the complement of the degree of change for the FDs of the BCS to address the second part of the second RQ. Therefore we counted the amount of features labelled as

CV, EV and PV and then calculated according to the formula in Section 2.3 δ_{CV_i} for every FD.

5.3 Results

Table 5.4 shows the evaluation results for the first experiment. The number of valid configurations of CS1 is $|CS_{CS1}| = 20$ and of the BCS is $|CS_{BCS-V1}| = 11616$. The number of valid and realisable configurations are expected to be $|RS^{GT}_{CS1}| = 4$ for CS1 and $|RS^{GT}_{BCS}| = 6528$ for the BCS.

Case Study		CS1	BCS
Valid Configurations ($\|CS\|$)		20	11616
Realisable Configurations	Ground Truth ($\|RS^{GT}\|$)	4	6528
	Our Results ($\|RS^{R}\|$)	4	6528
	Equality ($RS^{R} \equiv_{CS} RS^{GT}$)	True	True

Table 5.4: Quantitative results of our results compared to the GT for CS1 and BCS.

We computed $|RS^{R}_{CS1}| = 4$ valid and realisable configurations for CS1 and $|RS^{R}_{BCS}| = 6528$ for the BCS. The number of valid and realisable configurations for both CSs is equal to the expected number of the GT. We then compared the sets of configurations with the set of the GT and found an exact coverage ($RS^{R}_{CS1} \equiv_{CS} RS^{GT}_{CS1}$ and $RS^{R}_{BCS} \equiv_{CS} RS^{GT}_{BCS}$).

Table 5.5 shows the evaluation results for the second experiment, including the GT for Version 1.0 and 2.0 of the BCS and the BCS with deltas unapplied and deltas applied. The number of valid configurations for Version 1.0 of the BCS is $|CS_{BCS-V1}| = 11616$ and for Version 2.0 is $|CS_{BCS-V2}| \geq 115055$. The number of valid and realisable configurations are expected to be $|RS^{GT}_{BCS-\Delta}| = 6528$ for Version 1.0 of the BCS and $|RS^{GT}_{BCS-\Delta-Applied}| = 55296$ for Version 2.0 of the BCS.

Case Study		BCS-Δ	BCS-Δ-Applied
Valid Configurations ($\|CS\|$)		11616	≥ 115055
Realisable Configurations	Ground Truth ($\|RS^{GT}\|$)	6528	55296
	Our Results ($\|RS^{R}\|$)	6528	55296
	Equality ($RS^{R} \equiv_{CS} RS^{GT}$)	true	true

Table 5.5: Quantitative results of our results compared to the GT for unapplied and applied deltas to BCS.

We computed $|RS^{R}_{BCS-\Delta}| = 6528$ valid and realisable configurations for the BCS with unapplied deltas and $|RS^{R}_{BCS-\Delta-Applied}| = 55296$ for the BCS with applied deltas. The number of valid and realisable configurations for both versions is equal to the expected number of the GT. We then compared the sets of configurations with the set of the GT and found an exact coverage ($RS^{R}_{BCS-\Delta} \equiv_{CS} RS^{GT}_{BCS-\Delta}$ and $RS^{R}_{BCS-\Delta-Applied} \equiv_{CS} RS^{GT}_{BCS-\Delta-Applied}$).

Table 5.6 shows the share of CVs in the different versions of the FM of the BCS we calculated as the third experiment. The share of CVs, $\delta_{CV,i}$ is the complement of the degree of change.

i	Version	$\lvert G_i \rvert$	$\lvert PV_i \rvert$	$\lvert EV_i \rvert$	$\delta_{CV,i}$	degree of change	GT
0	Version 1	28	28	0	0%	100%	100%
1	Version 1.1	28	0	0	100%	0%	0%
2	Version 2	45	17	3	55.56%	44.44%	>20%
3	Version 3	46	1	1	95.65%	4.35%	<20%
4	Version 3.1	46	0	0	100%	0%	0%
5	Version 4	48	2	3	89.58%	10.42%	<20%
6	Version 5	49	1	1	95.92%	4.08%	<20%
7	Version 5.1	52	3	2	90.38%	9.62%	<20%

Table 5.6: Share of CVs of the BCS FMs

The proportion of CVs in the first version is 0%, which corresponds to a degree of change of 100%, as in the GT. The proportion of CVs in Version 1.1 and Version 3.1 is also in line with the GT at 100%. For Version 2 we can see from the table, almost half of the variations were not carried over, which is a high degree of change. Version 3.0 and Version 5.0 have a low degree of change of about 5%. Version 4.0 has a higher degree of change for than Version 3.0, but is also low at about 10%, as in Version 5.1.

5.4 Discussion

In the following we will answer the RQs and discuss our results.

RQ 1: Can the UCM be used to reproduce realisation analysis?

As shown in the section above, the UCM can be used to reproduce realisation analysis. We showed that the implemented UCM can be used for the computation of all valid and realisable features by reproducing the realisation analysis of Ochs [Och23]. We calculated the same sets of valid and realisable configurations using our unified system instances, leading to the conclusion that the UCM can be used to reproduce realisation, which answers our first RQ. We also showed the correctness of our UCM implementation with respect to the representation of FDs and solution space artefacts.

RQ 2: Can concepts of delta modelling be used to express variability in time in the extended UCM?

We integrated the concepts of delta modelling into the UCM to express variability in time. To show that our implementation is capable of semantically expressing the differences between two versions, we performed a realisation analysis on the two versions and compared the resulting sets of valid and realisable configurations with the GT. The exact coverage of the two sets leads to the conclusion that delta modelling concepts can be used to express variability in time in the UCM, which answers the second RQ. The third experiment showed that SGE variation types can be applied to FDs. Therefore, we calculated the degree of change between successive versions of FDs, which were in the range we expected for each version as our GT. This shows that the concepts of SGE can be applied to models outside the domain of mechanical engineering. Although we have applied these concepts to another model, they cannot be applied in the same way to the variation we have implemented in the UCM, because deltas in the UCM may contain whole sub-trees or modelling elements that are not comparable to FDs. Further work can start here to develop appropriate concepts.

5.5 Threats to Validity

In this section we will discuss threats to validity. Firstly we will discuss internal threats relating to the the design and conduct of the evaluation and secondly external threats relating to generalisability.

5.5.1 Internal Threats to Validity

Our evaluation primarily focused on comparing sets of valid and realisable configurations, focusing mainly on features, constraints, resource types, mappings, hardware components, deltas and versions. This evaluation does not provide full coverage of all implemented classes in the UCM, as certain elements such as `Software Component` and specific methods of the `Delta Operations` interface were not included. However, the classes and methods not included in the evaluation have very similar functionality to the classes covered by the evaluation. The included elements are therefore a representative sample of the system functionality, minimising the risk of malfunction in the elements not included. In addition, potential inconsistencies in the models and tools could lead to biases or errors. To counter this, we rely on existing and state of the art models and tools, which we only need to integrate or modify. This reduces risk as we do not have to develop everything from scratch. In addition, our reliance on the GT formed by Ochs [Och23] carries a significant risk, as any inaccuracies or discrepancies in this reference could affect our evaluation results. The risks may be migrated for the GT formed by Ochs [Och23] for the following reasons. The set of valid and realisable configurations is always smaller than the set of valid configurations. This is by design, because the CSs were created in such a way that not every configuration is realisable even if it is valid. Also, the rule of which configurations are not realisable where made as simple as possible to allow manual verification. The valid configurations, unlike the valid and realisable configurations, were computed by the FeatureIDE which is a widely used tool. We also minimised the risk of errors in our model by using bi-directional parsing from FeatureIDE FDs into a unified system instance. This parsing ensures that no data is lost in the parsing process and that the features and constraints work as intended.

5.5.2 External Threats to Validity

External threats to validity could limit the generalisability of the research findings. Firstly, the focus on the BCS-CS means that the findings may not be generalisable to other contexts. Focusing primarily on one CS is not representative of other PLs. However, we used the BCS-CS in particular because, according to Müller et al. [MLD+09], it was developed with experts to mimic a real PL, which increases the generalisability of the results of our experiment. Another threat to external validity are scalability issues, as the analysis is limited to computing the valid and realisable configurations for Version 1.0 and 2.0 of the BCS-CS, with later versions shown as too complex to compute in an acceptable time. This limitation can be explained by combinatorial explosion, where the number of possible combinations grows exponentially. Although we have not been able to compute the valid and realisable configurations for later versions, we have been able to model them as a unified system and, as demonstrated with Version 1.0 and Version 2.0 of the BCS-CS, given enough time it would be possible to compute the results for later versions. However, our work focuses on modelling and replicating existing results and since we have shown that our work can be used to replicate existing CSs and we have not changed any analysis algorithms, as long as realisation analysis is possible with other methods, we can also perform it with our method.

6. Related Work

This chapter presents concepts and models that are relevant to our work. We analysed concepts that express variability in time or space within the domains of software engineering and mechanical engineering and approaches that introduce metamodels related to the UCM.

Concepts to express Variability in Time and Space

Apel et al. [ABKS13] divide PL into problem and solution space and introduce the FM as a model that allows the management of variability in space by the FD as a graphical representation. The SGE is a concept of expressing variability in time in the domain of mechanical engineering by expressing successor generations through different types of variations, starting from a previous generation, according to Albers et al. [ABR17]. The concept of delta modelling express the change operations to a system instance, this can be applied either to configurations to manage variability in space according to Schaefer [Sch10] or to versions to manage variability in time according to Lity et al. [LKS16]. These concepts and models provide solutions for expressing variability in time or space. However, these preceding approaches cannot cover variability in time and space at the same time as in the case of the FM and SGE, or cannot provide realisation analysis because they do not cover problem and solution space. We integrated the expression of variability in time by adopting the idea of FD and integrated the SGE variation types through time deltas, which we also used to express variability in time. This allows us to cover variability in time and space and to cover problem and solution space through the unified approach of the UCM.

Combined Variability Models

The UCM is a metamodel proposed by Ananieva et al. [AGK+22] to express variability in time and space in PLs. The solution space of this model was extended by Wittler et al. [WKR22] and Ochs [Och23] with a more precise model that allow analyses such as realisation analysis. Our work replicates the realisation analysis by implementing the UCM and performing the analysis directly on an instance of the UCM, which was previously performed outside the boundaries of the model.

Consistency Models

Atkinson et al. [ASB10] proposed the approach of a Single Underlying Model (SUM) where a single, unified model serves as the foundation for multiple perspectives in view-based development. Burger [Bur13] introduces Virtual SUM (VSUM) as a unified metamodel which is constructed by defining consistency relations between involved metamodels. The UCM we developed can be seen as a SUM, since the UCM is the foundation for multiple different perspectives we combine, as variants through the integration of the FD or versions through delta modelling. The UCM already includes approaches for the mapping between features and the concrete realisation in the solution space by software and hardware components, but lacks a concrete realisation for maintaining constancy between features and their realisation in software and hardware components.

7. Conclusion and Outlook

This thesis addresses the description and management of variability in space and time of a generic PL in a unified model. The UCM is a model that unifies the problem and solution space and is able to describe variability in space and time. FDs enable the management of variability in space and the SGE is used in the domain of mechanical engineering to manage variability in time. We integrate these concepts into the UCM to enable the management of variability in time and space.

We structured this thesis along the following SGs we introduced in Chapter 1:

SG1 Implementation of the UCM in the EMF

SG2 Application study of the implementation (SG 1) based on the BCS with regard to realisability

SG3 Development of a concept for the integration of the SGEM into the UCM with regard to variability in time

SG4 Further development of the BCS by adding evolution

We addressed SG 1 in Section 3.1 and Section 4.1. Therefore, we extended the UCM to allow us to manage variability in time. We then implemented the UCM in the EMF and also implemented adapters to import and export from and to FeatureIDE FD serialisations. We also added adapters to import resource types and resource provisions from CSV files as used by Ochs [Och23]. These adapted files can then be saved as serialised instance of UCM to allow the application of analysis techniques such as realisation analysis.

SG 2 was addressed in Chapter 5 and Section 4.1. We implemented the tool support for the replication study of the solution space analysis proposed by Ochs [Och23], where the realisation of valid configurations is addressed. We used the serialisation of the UCM implemented in SG 1 as input for the modified analysis method. In the evaluation, we investigated whether our implementation of the UCM is able to reproduce existing realisation analysis results from Ochs [Och23]. We applied the realisation analysis method to two CSs and then compared our results with the GT, provided by Ochs [Och23]. We found exact coverage in the sets of valid and realisable configurations of both CSs, thus proving the applicability of our implementation of the UCM.

SG 3 was addressed in Section 3.2, where we used delta modelling to describe changes between two versions of a PL. We integrate this concept into the UCM, which enables it to express variability in time. The deltas, designed in the UCM, can be labelled with the SGE variation types, allowing the description of variation types in cross-domain PLs.

We addressed SG 4 in Section 4.2 and Chapter 5, where we applied our implementation of variability in time in the UCM to the BCS-CS. Therefore, we extended the BCS in terms of evolution and solution space artefacts, which allowed us to perform realisability analysis on the different versions of the BCS. We used this realisability analysis to evaluate whether the implementation of the UCM was able to semantically express differences between two versions with our concept of variability in time. We found an exact coverage of the sets of valid and realisable configurations, which we expressed by delta modelling and compared with the GT. Thus, we showed that our concept of variability in time in the UCM is capable of correctly specifying all changes between two versions of the subject system.

At the current state of our work, if a mapping points to a feature and the feature is removed, the model is inconsistent because the mappings feature reference points to a removed feature but is not itself removed. To circumvent such inconsistencies structure-preserving deltas could be added, which enable cascading deletions.
With the extended UCM, we provide a metamodel that covers the management of variability in space and time in both the problem and solution space. This opens up opportunities for further development of tools for joint analysing of the problem and solution space, such as combined realisation analysis.
The SGE variation types in our work are applied only to features, while originally in the domain of mechanical engineering the SGE variation types were applied to hardware components. Since we label deltas with variation types and deltas can be applied to arbitrary elements in a metamodel, other elements could be considered for change analysis in future work.

Bibliography

[AA07] Alfred V. Aho and Alfred V. Aho, editors. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd ed edition, 2007. OCLC: ocm70775643.

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

[ABR17] Albert Albers, Nikola Bursac, and Simon Rapp. PGE – Produktgenerationsentwicklung am Beispiel des Zweimassenschwungrads. *Forschung im Ingenieurwesen*, 81(1):13–31, March 2017.

[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, May 2022.

[ASB10] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Orthographic Software Modeling: A Practical Approach to View-Based Development. In Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 206–219, Berlin, Heidelberg, 2010. Springer.

[Bur13] Erik Johannes Burger. Flexible views for view-based model-driven development. In *Proceedings of the 18th international doctoral symposium on Components and architecture*, pages 25–30, Vancouver British Columbia Canada, June 2013. ACM.

[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*, VaMoS '20, pages 1–10, New York, NY, USA, February 2020. Association for Computing Machinery.

[LKS16] Sascha Lity, Matthias Kowal, and Ina Schaefer. Higher-order delta modeling for software product line evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, pages 39–48, Amsterdam Netherlands, October 2016. ACM.

[MLD+09] Tobias Müller, Malte Lochau, Stefan Detering, Falko Saust, Henning Garbers, Lukas Märtin, Thomas Form, and Ursula Goltz. A comprehensive description of a model-based, continous development process for AUTOSAR systems with integrated quality assurance. 2009.

[NC13] Linda M Northrop and Paul C Clements. A Framework for Software Product Line Practice, Version 5.0. 2013.

[Och23] Philip Ochs. Combined Modeling of Software and Hardware with Versions and Variants, 2023.

[Sch10] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. pages 85–92, January 2010.

[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*, volume B of *SPLC '22*, pages 94–98, New York, NY, USA, September 2022. Association for Computing Machinery.

A. Appendix

A.1 Body Comfort System-Case Study

A.1.1 Version 1.0

See Section 4.2.

A.1.2 Version 1.1 - Safety

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 25$ $rp_7^0 = 16$
`security-hardware`(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$

Table A.1: Resource Provisions of Version 1.1

A.1.3 Version 2.0 - Wiper System

See Section 4.2.

A.1.4 Version 3.0 - Electric Seat Adjustment Based on Key IDs

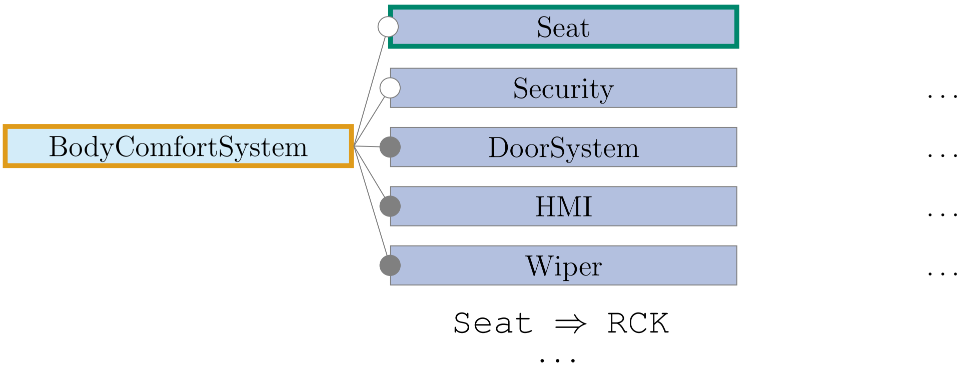


Figure A.1: FM of BCS Version 3.0

ID	isAdditive	isExclusive	boundary	Unit	Description
11	True	True	LOWER		no. Motors
12	True	False	LOWER	KB	Long Term Memory

Table A.2: Resource Types added in Version 3.0

seat-controller (sw_5)

Feature	Software Component sw_i	Resource Demands rd_k^i
Seat	sw_5	$rd_{11}^5 = 1$ $rd_{12}^5 = 2$ $rd_7^5 = 5$

Table A.3: Resource Demands added in Version 3.0

Hardware Component (hw_j)	Provisions (rp_k^j)
infotainment-hardware(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 35$ $rp_7^0 = 16$ $rp_{11}^0 = 1$ $rp_{12}^0 = 1$
security-hardware(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
safety-hardware(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$ $rp_8^2 = 1$ $rp_9^2 = 1$ $rp_{10}^2 = 1$

Table A.4: Resource Provisions of Version 3.0

A.1.5 Version 3.1 - Safer Passengers 2

ID	isAdditive	isExclusive	boundary	Unit	Description
13	True	False	UPPER	ms	Response Time

Table A.5: Resource Types added in Version 3.1

Feature	Software Component sw_i	Resource Demands rd_k^i
FP	sw_2	$rd_6^2 = 1$ $rd_{13}^2 = 2$

Table A.6: Resource Demands added in Version 3.1

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 12$ $rp_1^0 = 35$ $rp_7^0 = 16$ $rp_{11}^0 = 1$ $rp_{12}^0 = 1$
`security-hardware`(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$ $rp_8^2 = 1$ $rp_9^2 = 1$ $rp_{10}^2 = 1$ $rp_{13}^2 = 2$
`security-hardware-2`(hw_3)	$rp_5^3 = 1$

Table A.7: Resource Provisions added in Version 3.1

A.1.6 Version 4.0 - Heatable Windows

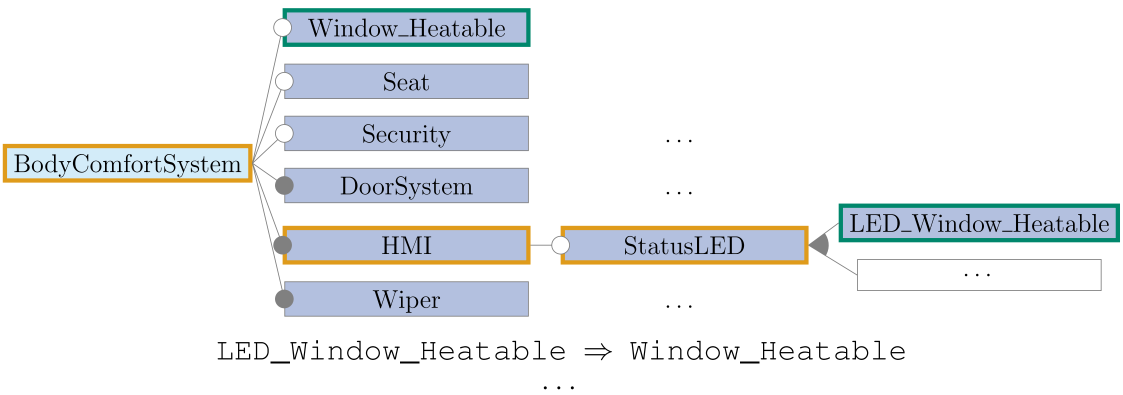


Figure A.2: FM of BCS Version 4.0

`windows-heat-controller` (sw_6)

Feature	Software Component sw_i	Resource Demands rd_k^i
LED_Heatable	sw_0	$rd_0^0 = 1$
Window_Heatable	sw_6	$rd_9^6 = 1$

Table A.8: Resource Demands added in Version 4.0

A.1.7 Version 5.0 - Automatic Headlights

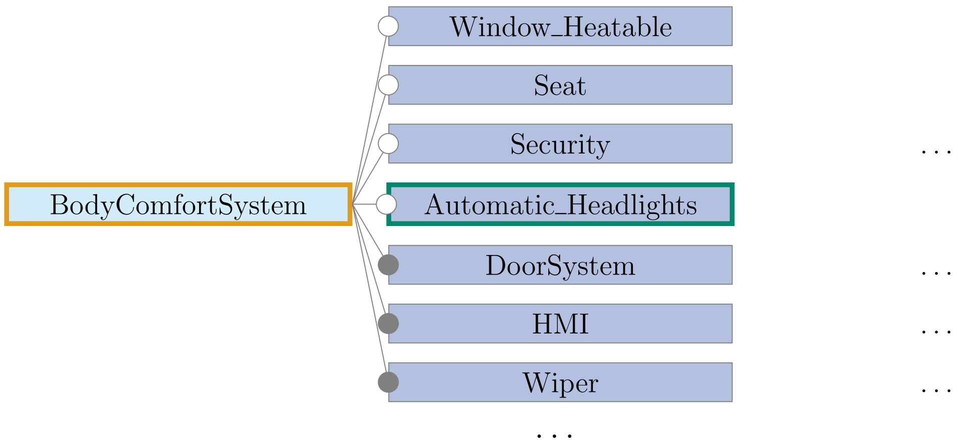


Figure A.3: FM of BCS Version 5.0

ID	isAdditive	isExclusive	boundary	Unit	Description
14	True	True	LOWER		no. Parking Lights
15	True	True	LOWER		no. Daytime Running Lights
16	True	True	LOWER		no. Low Beam
17	True	True	LOWER		no. High Beam
18	True	False	LOWER		no. Ambient Light Sensor
19	True	False	LOWER		no. Front Proximity Sensor

Table A.9: Resource Types added in Version 5.0

headlight-controller (sw_7)

Feature	Software Component sw_i	Resource Demands rd_k^i
Automatic_Headlights	sw_7	$rd_{14}^7 = 1$ $rd_{15}^7 = 1$ $rd_{16}^7 = 1$ $rd_{17}^7 = 1$ $rd_{18}^7 = 1$ $rd_{19}^7 = 1$

Table A.10: Resource Demands added in Version 5.0

Hardware Component (hw_j)	Provisions (rp_k^j)
`headlight-hardware`(hw_4)	$rp_{14}^4 = 1$ $rp_{15}^4 = 1$ $rp_{16}^4 = 1$ $rp_{17}^4 = 1$ $rp_{18}^4 = 1$ $rp_{19}^4 = 1$

Table A.11: Resource Provisions added in Version 5.0

A.1.8 Version 5.1 - Automatic Headlights Decisions

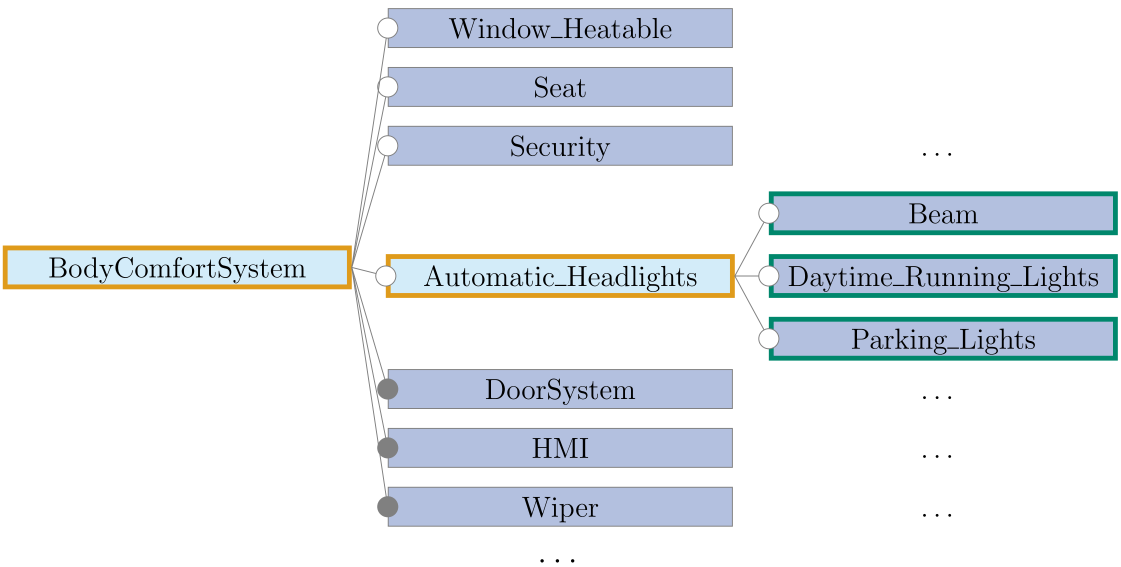


Figure A.4: FM of Version 5.1 BCS

ID	isAdditive	isExclusive	boundary	Unit	Description
~~14~~	~~`True`~~	~~`True`~~	~~`LOWER`~~		~~no. Parking Lights~~
~~15~~	~~`True`~~	~~`True`~~	~~`LOWER`~~		~~no. Daytime Running Lights~~
~~16~~	~~`True`~~	~~`True`~~	~~`LOWER`~~		~~no. Low Beam~~
~~17~~	~~`True`~~	~~`True`~~	~~`LOWER`~~		~~no. High Beam~~
~~18~~ 14	`True`	`False`	`LOWER`		no. Ambient Light Sensor
~~19~~ 15	`True`	`False`	`LOWER`		no. Front Proximity Sensor

Table A.12: Resource Types added and removed in Version 5.1

Feature	Software Component sw_i	Resource Demands rd_k^i
Parking_Lights	sw_7	$rd_1^7 = 5$ $rd_{14}^7 = 1$
Beam	sw_7	$rd_1^7 = 55$ $rd_{14}^7 = 1$ $rd_{15}^7 = 1$
Daytime_Running_Lights	sw_7	$rd_1^7 = 5$ $rd_{14}^7 = 1$ $rd_{15}^7 = 1$
Automatic_Headlights	sw_7	

Table A.13: Resource Demands added and removed in Version 5.1

Hardware Component (hw_j)	Provisions (rp_k^j)
`headlight-hardware`(hw_4)	$rp_7^4 = 100$ $rp_{14}^4 = 1$ $rp_{15}^4 = 1$

Table A.14: Resource Provisions modified in Version 5.1

A.2 Modified Unified Conceptual Model Full

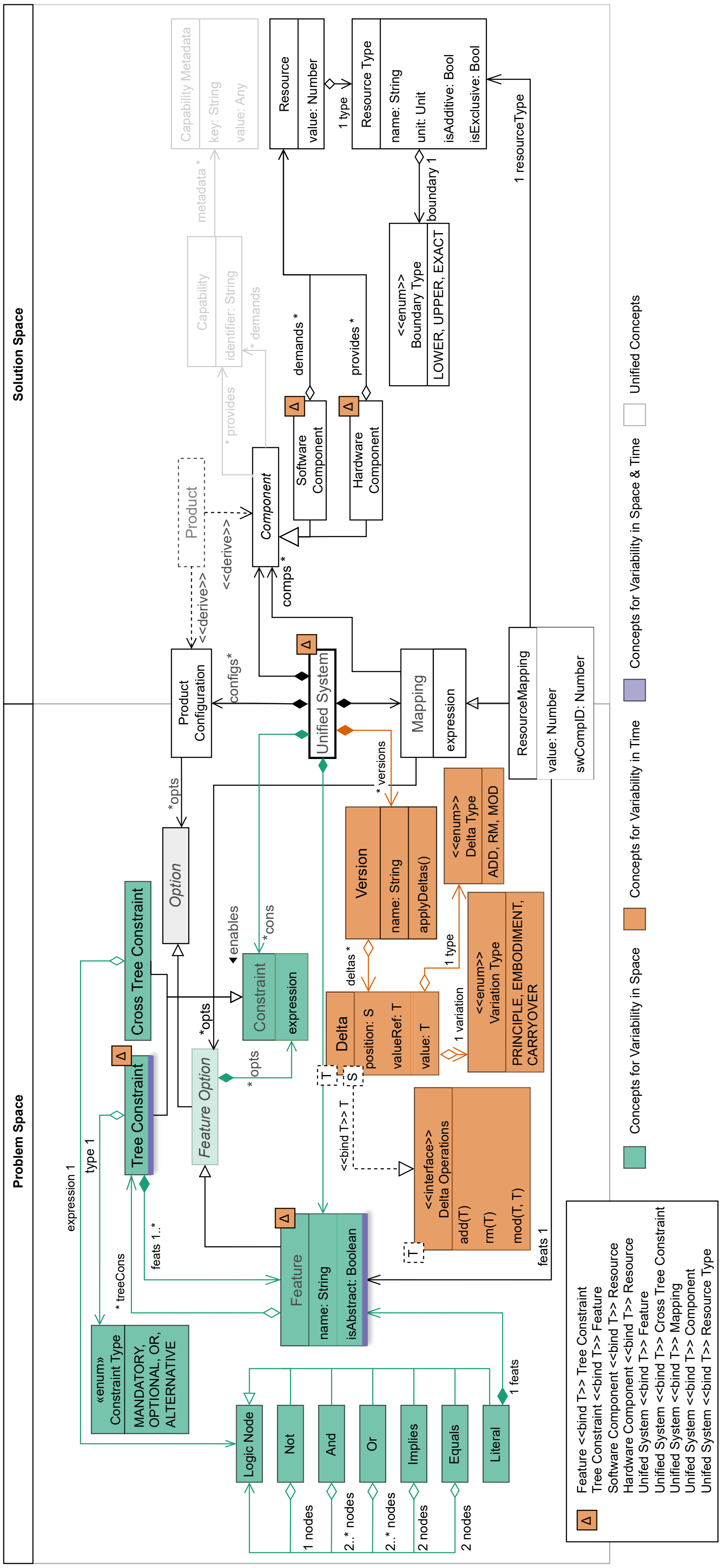


Figure A.5: Full Extended UCM